



(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:

27.05.1998 Bulletin 1998/22

(51) Int. Cl.⁶: G06F 9/46, G06F 9/44

(21) Application number: 97110839.4

(22) Date of filing: 01.07.1997

(84) Designated Contracting States:
AT BE CH DE DK ES FI FR GB GR IE IT LI LU MC
NL PT SE(72) Inventor: Gish, Sherri L.
Mountain View, California 94043 (US)

(30) Priority: 01.07.1996 US 675253

(74) Representative:
Kindermann, Manfred
Patentanwalt,
Sperberweg 29
71032 Böblingen (DE)(71) Applicant:
SUN MICROSYSTEMS, INC.
Mountain View, California 94043-1100 (US)

(54) Object-oriented system, method and article of manufacture for a client-server event driver message framework in an enterprise computing framework system

(57) An enterprise computing manager in which an application is composed of a client (front end) program which communicates utilizing a network with a server (back end) program. The client and server programs are loosely coupled and exchange information using the network. The client program is composed of a User Interface (UI) and an object-oriented framework (Presentation Engine (PE) framework). The UI exchanges data messages with the framework. The framework is designed to handle two types of messages: (1) from the UI, and (2) from the server (back end) program via the network. The framework includes a component, the mediator which manages messages coming into and going out of the framework. The system includes software for a client computer, a server computer and a network for connecting the client computer to the server

computer which utilize an execution framework code segment configured to couple the server computer and the client computer via the network, by a plurality of client computer code segments resident on the server, each for transmission over the network to a client computer to initiate coupling; and a plurality of server computer code segments resident on the server which execute on the server in response to initiation of coupling via the network with a particular client utilizing the transmitted client computer code segment for communicating via a particular communication protocol. A mediator state machine is utilized to parse various message types and route the messages to appropriate parts of the execution framework for further processing.

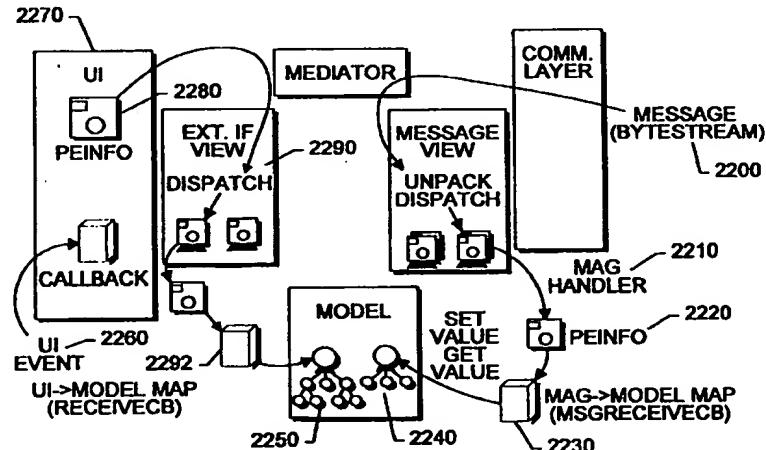


FIG.-22

BEST AVAILABLE COPY

Description**COPYRIGHT NOTIFICATION**

5 Portions of this patent application contain materials that are subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document, or the patent disclosure, as it appears in the Patent and Trademark Office.

Field of the Invention

10 This invention generally relates to improvements in computer systems and, more particularly, to operating system software for managing enterprise computing in a network user interface.

Background of the Invention

15 One of the most important aspects of a modern computing system is the interface between the human user and the machine. The earliest and most popular type of interface was text based; a user communicated with the machine by typing text characters on a keyboard and the machine communicated with the user by displaying text characters on a display screen. More recently, graphic user interfaces have become popular where the machine communicates with 20 a user by displaying graphics, including text and pictures, on a display screen and the user communicates with the machine both by typing in textual commands and by manipulating the displayed pictures with a pointing device, such as a mouse.

25 Many modern computer systems operate with a graphic user interface called a window environment. In a typical window environment, the graphical display portrayed on the display screen is arranged to resemble the surface of an electronic "desktop" and each application program running on the computer is represented as one or more electronic "paper sheets" displayed in rectangular regions of the screen called "windows".

30 Each window region generally displays information which is generated by the associated application program and there may be several window regions simultaneously present on the desktop, each representing information generated by a different application program. An application program presents information to the user through each window by drawing or "painting" images, graphics or text within the window region. The user, in turn, communicates with the application by "pointing at" objects in the window region with a cursor which is controlled by a pointing device and manipulating or moving the objects and also by typing information into the keyboard. The window regions may also be moved around on the display screen and changed in size and appearance so that the user can arrange the desktop in a convenient manner.

35 Each of the window regions also typically includes a number of standard graphical objects such as sizing boxes, buttons and scroll bars. These features represent user interface devices that the user can point at with the cursor to select and manipulate. When the devices are selected or manipulated, the underlying application program is informed, via the window system, that the control has been manipulated by the user.

40 In general, the window environment described above is part of the computer operating system. The operating system also typically includes a collection of utility programs that enable the computer system to perform basic operations, such as storing and retrieving information on a disc memory, communicating with a network and performing file operations including the creation, naming and renaming of files and, in some cases, performing diagnostic operations in order to discover or recover from malfunctions.

45 The last part of the computing system is the "application program" which interacts with the operating system to provide much higher level functionality, perform a specific task and provide a direct interface with the user. The application program typically makes use of operating system functions by sending out series of task commands to the operating system which then performs a requested task. For example, the application program may request that the operating system store particular information on the computer disc memory or display information on the video display.

50 Figure 1 is a schematic illustration of a typical prior art computer system utilizing both an application program and an operating system. The computer system is schematically represented by box 100, the application is represented by box 102 and the operating system by box 106. The previously-described interaction between the application program 102 and the operating system 106 is illustrated schematically by arrow 104. This dual program system is used on many types of computer systems ranging from main frames to personal computers.

55 The method for handling screen displays varies from computer to computer and, in this regard, Figure 1 represents a prior art personal computer system. In order to provide screen displays, application program 102 generally stores information to be displayed (the storing operation is shown schematically by arrow 108) into a screen buffer 110. Under control of various hardware and software in the system the contents of the screen buffer 110 are read out of the buffer and provided, as indicated schematically by arrow 114, to a display adapter 112. The display adapter 112 contains

hardware and software (sometimes in the form of firmware) which converts the information in screen buffer 110 to a form which can be used to drive the display monitor 118 which is connected to display adapter 112 by cable 116.

The prior art configuration shown in Figure 1 generally works well in a system where a single application program 102 is running at any given time. This simple system works properly because the single application program 102 can write information into any area of the entire screen buffer area 110 without causing a display problem. However, if the configuration shown in Figure 1 is used in a computer system where more than one application program 102 can be operational at the same time (for example, a "multi-tasking" computer system) display problems can arise. More particularly, if each application program has access to the entire screen buffer 110, in the absence of some direct communication between applications, one application may overwrite a portion of the screen buffer which is being used by another application, thereby causing the display generated by one application to be overwritten by the display generated by the other application.

Accordingly, mechanisms were developed to coordinate the operation of the application programs to ensure that each application program was confined to only a portion of the screen buffer thereby separating the other displays. This coordination became complicated in systems where windows were allowed to "overlap" on the screen display. When the screen display is arranged so that windows appear to "overlap", a window which appears on the screen in "front" of another window covers and obscures part of the underlying window. Thus, except for the foremost window, only part of the underlying windows may be drawn on the screen and be "visible" at any given time. Further, because the windows can be moved or resized by the user, the portion of each window which is visible changes as other windows are moved or resized. Thus, the portion of the screen buffer which is assigned to each application window also changes as windows from other applications are moved or resized.

In order to efficiently manage the changes to the screen buffer necessary to accommodate rapid screen changes caused by moving or resizing windows, the prior art computer arrangement shown in Figure 1 was modified as shown in Figure 2. In this new arrangement computer system 200 is controlled by one or more application programs, of which programs 202 and 216 are shown, which programs may be running simultaneously in the computer system. Each of the programs interfaces with the operating system 204 as illustrated schematically by arrows 206 and 220. However, in order to display information on the display screen, application programs 202 and 216 send display information to a central window manager program 218 located in the operating system 204. The window manager program 218, in turn, interfaces directly with the screen buffer 210 as illustrated schematically by arrow 208. The contents of screen buffer 210 are provided, as indicated by arrow 212, to a display adapter 214 which is connected by a cable 222 to a display monitor 224.

In such a system, the window manager 218 is generally responsible for maintaining all of the window displays that the user views during operation of the application programs. Since the window manager 218 is in communication with all application programs, it can coordinate between applications to insure that window displays do not overlap. Consequently, it is generally the task of the window manager to keep track of the location and size of the window and the window areas which must be drawn and redrawn as windows are moved.

The window manager 218 receives display requests from each of the applications 202 and 216. However, since only the window manager 218 interfaces with the screen buffer 210, it can allocate respective areas of the screen buffer 210 for each application and insure that no application erroneously overwrites the display generated by another application. There are a number of different window environments commercially available which utilize the arrangement illustrated in Figure 2. These include the X/Window Operating environment, the WINDOWS graphical user interface developed by the Microsoft Corporation and the OS/2 Presentation Manager developed by the International Business Machines Corporation, and the Macintosh OS, developed by Apple Computer Corporation.

Each of these window environments has its own internal software architecture, but the architectures can all be classified by using a multi-layer model similar to the multi-layer models used to describe computer network software. A typical multi-layer model includes the following layers:

50

- User Interface
- Window Manager
- Resource Control and Communication
- Component Driver Software
- Computer Hardware

where the term "window environment" refers to all of the above layers taken together.

The lowest or computer hardware level includes the basic computer and associated input and output devices including display monitors, keyboards, pointing devices, such as mice or trackballs, and other standard components, including printers and disc drives. The next or "component driver software" level consists of device-dependent software that generates the commands and signals necessary to operate the various hardware components. The resource control and communication layer interfaces with the component drivers and includes software routines which allocate

resources, communicate between applications and multiplex communications generated by the higher layers to the underlying layers. The window manager handles the user interface to basic window operations, such as moving and resizing windows, activating or inactivating windows and redrawing and repainting windows. The final user interface layer provides high level facilities that implement the various controls (buttons, sliders, boxes and other controls) that application programs use to develop a complete user interface.

5 Although the arrangement shown in Figure 2 solves the display screen interference problem, it suffers from the drawback that the window manager 218 must process the screen display requests generated by all of the application programs. Since the requests can only be processed serially, the requests are queued for presentation to the window manager before each request is processed to generate a display on terminal 224. In a display where many windows are
10 present simultaneously on the screen, the window manager 218 can easily become a "bottleneck" for display information and prevent rapid changes by of the display by the application programs 202 and 216. A delay in the redrawing of the screen when windows are moved or repositioned by the user often manifests itself by the appearance that the windows are being constructed in a piecemeal fashion which becomes annoying and detracts from the operation of the system.

15 This problem becomes even more accentuated in a client-server environment where many applications are all in contention for very limited resources. The Internet has permeated the workplace as a communication medium of choice. Since the internet is accessible from almost any point in a typical business enterprise a new buzzword has evolved from the term enterprise computer into an "Enterprise" computer. Enterprise is a concatenation of internet and enterprise.

20 In today's client server enterprises, applications that exist in current client server enterprises are not really built to be managed since they are architected for distributed system environments. New systems are also required to be evolutionary, not revolutionary. Redesign of current systems that require significant expense need to be avoided.

25 A system is required that allows a user to create manageable applications, that can be readily deployed, installed on a variety of platforms, and configured to facilitate partitioning them on clients versus servers and administer the applications once they're running. Systems don't always break because of failure, errors, or bugs, they sometimes break because the enterprise itself is complicated and somebody does something unexpected somewhere which will bring the whole system down. When the system does come down, then a system administrator must be able to readily identify the problems, and deal with them in an effective manner so that a business doesn't stop functioning when one of these unforeseen events happens.

30 The application should be designed based on domain requirements, so it is independent of any platform underneath, and fits more with how commercial developers work. In the commercial world, the development process isn't that important. The regular employees are not applications developers or programmers. Companies usually hire such work out; they get consultants to do that kind of work. Depending on the company and what they want done, it usually hires a consulting firm, individual consultants, or smaller groups of consultants to come in, help it develop an application.
35 Their goal is the end application, which must be maintained. The company configures it, evolves it, and grows it. To allow for modification, the development task must be modular to allow different groups of people working on different parts of an application, without requiring any one group to understand every detail of the whole application of the enterprise.

40 The second criterion requires minimal extra knowledge, burden or sophistication on the part of the people developing the system. Most companies do not desire to have their business hinge on a single individual. Rather, they desire to have people who are primarily domain experts who can work with well-understood tools to produce a application matching company requirements quickly without making special demands on the company.

Summary of the Invention

45 The foregoing problems are overcome in an illustrative embodiment of the invention in which an application is composed of a client (front end) program which communicates utilizing a network with a server (back end) program. The client and server programs are loosely coupled and exchange information using the network. The client program is composed of a User Interface (UI) and an object-oriented framework (Presentation Engine (PE) framework). The UI exchanges data messages with the framework. The framework is designed to handle two types of messages: (1) from the UI, and (2) from the server (back end) program via the network. The framework includes a component, the mediator which manages messages coming into and going out of the framework.

50 A distributed computer system is disclosed with software for a client computer, a server computer and a network for connecting the client computer to the server computer which utilize an execution framework code segment configured to couple the server computer and the client computer via the network, by a plurality of client computer code segments resident on the server, each for transmission over the network to a client computer to initiate coupling; and a plurality of server computer code segments resident on the server which execute on the server in response to initiation of coupling via the network with a particular client utilizing the transmitted client computer code segment for communi-

cating via a particular communication protocol. A mediator state machine is utilized to parse various message types and route the messages to appropriate parts of the execution framework for further processing. The presentation engine manages message processing utilizing the presentation engine framework and the user interface.

5 **Brief Description of the Drawings**

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings, in which:

10 Figure 1 is a schematic block diagram of a prior art computer system showing the relationship of the application program, the operating system, the screen buffer and, the display monitor;

Figure 2 is a schematic block diagram of a modification of the prior art system shown in Figure 1 which allows several application programs running simultaneously to generate screen displays;

15 Figure 3 is a block schematic diagram of a computer system for example, a personal computer system on which the inventive object oriented window manager operates;

Figure 4 is a block diagram in accordance with a preferred embodiment;

20 Figure 5 illustrates how a preferred embodiment leverages Java to facilitate the establishment and implementation of sever-centric policies;

Figure 6 illustrates the processing associated with application startup in accordance with a preferred embodiment;

25 Figure 7 illustrates the three fundamental components of an application in accordance with a preferred embodiment;

30 Figure 8 illustrates the migration of an existing client-server application to one supported by a preferred embodiment;

Figure 9 is a block diagram illustrating a PE 960 in accordance with a preferred embodiment;

35 Figure 10 is a block diagram of a prior art client server architecture in accordance with a preferred embodiment;

Figure 11 illustrates an application in accordance with an alternate embodiment;

40 Figure 12 illustrates a server 1210 establishing contact with a client 1200 in accordance with an alternate embodiment;

Figure 13 illustrates a loosely coupled client - server application in accordance with an alternate embodiment;

45 Figure 14 illustrates the system integration task necessary to develop an application 1430 in accordance with a preferred embodiment;

Figure 15 is a block diagram illustrating the modular design of a client application in accordance with a preferred embodiment;

50 Figure 16 is a block diagram of a framework in accordance with an alternate embodiment;

Figure 17 illustrates the basic building blocks in accordance with an alternate embodiment;

55 Figure 18 is a block diagram highlighting the steps utilized to extend the framework in accordance with a preferred embodiment;

Figure 19 is an illustration of a PE Object in accordance with a preferred embodiment;

Figure 20 is an illustration of a PE Event Handler 2000 used by Views to handle incoming messages and UI events

in accordance with a preferred embodiment;

Figure 21 illustrates a PEInfo object 2100 data in accordance with a preferred embodiment;

5 Figure 22 illustrates incoming message flow to a model in accordance with a preferred embodiment;

Figure 23 illustrates incoming messages mapping a UI to a Model in accordance with a preferred embodiment;

10 Figure 24 illustrates outgoing messages mapping a model to messages in accordance with a preferred embodiment;

Figure 25 illustrates outgoing messages mapping a model to a UI in accordance with a preferred embodiment;

15 Figure 26 illustrates the steps associated with launching an application URL in accordance with a preferred embodiment;

Figure 27 describes the forms of a Presentation Engine, as an abstract Java class, a template for development, and an executable component in an application in accordance with a preferred embodiment;

20 Figure 28 describes the functions developers must fill in using the server program template in accordance with a preferred embodiment;

Figure 29 illustrates Server Properties in accordance with a preferred embodiment; and

25 Figure 30 is a table of client and serverside exceptions in accordance with a preferred embodiment.

Detailed Description

The invention is preferably practiced in the context of an operating system resident on a computer such as a SUN, 30 IBM PS/2 or Apple Macintosh computer. A representative hardware environment is depicted in Figure 3, which illustrates a typical hardware configuration of a computer 300 in accordance with the subject invention. The computer 300 is controlled by a central processing unit 302 (which may be a conventional microprocessor) and a number of other units, all interconnected via a system bus 308, are provided to accomplish specific tasks. Although a particular computer may only have some of the units illustrated in Figure 3, or may have additional components not shown, most computers will include at least the units shown.

35 Specifically, computer 300 shown in Figure 3 includes a random access memory (RAM) 306 for temporary storage of information, a read only memory (ROM) 304 for permanent storage of the computer's configuration and basic operating commands and an input/output (I/O) adapter 310 for connecting peripheral devices such as a disk unit 313 and printer 314 to the bus 308, via cables 315 and 312, respectively. A user interface adapter 316 is also provided for connecting input devices, such as a keyboard 320, and other known interface devices including mice, speakers and microphones to the bus 308. Visual output is provided by a display adapter 318 which connects the bus 308 to a display device 322, such as a video monitor. The computer has resident thereon and is controlled and coordinated by operating system software such as the SUN Solaris or JavaOS operating system.

40 In a preferred embodiment, the invention is implemented in the C++ programming language using object-oriented programming techniques. C++ is a compiled language, that is, programs are written in a human-readable script and this script is then provided to another program called a compiler which generates a machine-readable numeric code that can be loaded into, and directly executed by, a computer. As described below, the C++ language has certain characteristics which allow a software developer to easily use programs written by others while still providing a great deal of control over the reuse of programs to prevent their destruction or improper use. The C++ language is well-known and many 45 articles and texts are available which describe the language in detail. In addition, C++ compilers are commercially available from several vendors including Borland International, Inc. and Microsoft Corporation. Accordingly, for reasons of clarity, the details of the C++ language and the operation of the C++ compiler will not be discussed further in detail herein.

45 As will be understood by those skilled in the art, Object-Oriented Programming (OOP) techniques involve the definition, creation, use and destruction of "objects". These objects are software entities comprising data elements and routines, or functions, which manipulate the data elements. The data and related functions are treated by the software as an entity and can be created, used and deleted as if they were a single item. Together, the data and functions enable objects to model virtually any real-world entity in terms of its characteristics, which can be represented by the data ele-

ments, and its behavior, which can be represented by its data manipulation functions. In this way, objects can model concrete things like people and computers, and they can also model abstract concepts like numbers or geometrical designs.

Objects are defined by creating "classes" which are not objects themselves, but which act as templates that instruct the compiler how to construct the actual object. A class may, for example, specify the number and type of data variables and the steps involved in the functions which manipulate the data. An object is actually created in the program by means of a special function called a constructor which uses the corresponding class definition and additional information, such as arguments provided during object creation, to construct the object. Likewise objects are destroyed by a special function called a destructor. Objects may be used by using their data and invoking their functions.

The principle benefits of object-oriented programming techniques arise out of three basic principles; encapsulation, polymorphism and inheritance. More specifically, objects can be designed to hide, or encapsulate, all, or a portion of, the internal data structure and the internal functions. More particularly, during program design, a program developer can define objects in which all or some of the data variables and all or some of the related functions are considered "private" or for use only by the object itself. Other data or functions can be declared "public" or available for use by other programs. Access to the private variables by other programs can be controlled by defining public functions for an object which access the object's private data. The public functions form a controlled and consistent interface between the private data and the "outside" world. Any attempt to write program code which directly accesses the private variables causes the compiler to generate an error during program compilation which error stops the compilation process and prevents the program from being run.

Polymorphism is a concept which allows objects and functions which have the same overall format but which work with different data, to function differently in order to produce consistent results. For example, an addition function may be defined as variable A plus variable B ($A+B$) and this same format can be used whether the A and B are numbers, characters or dollars and cents. However, the actual program code which performs the addition may differ widely depending on the type of variables that comprise A and B. Polymorphism allows three separate function definitions to be written, one for each type of variable (numbers, characters and dollars). After the functions have been defined, a program can later refer to the addition function by its common format ($A+B$) and, during compilation, the C++ compiler will determine which of the three functions is actually being used by examining the variable types. The compiler will then substitute the proper function code. Polymorphism allows similar functions which produce analogous results to be "grouped" in the program source code to produce a more logical and clear program flow.

The third principle which underlies object-oriented programming is inheritance, which allows program developers to easily reuse pre-existing programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to declare classes (and the objects which are later created from them) as related. Specifically, classes may be designated as subclasses of other base classes. A subclass "inherits" and has access to all of the public functions of its base classes just as if these function appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions or may modify some or all of its inherited functions merely by defining a new function with the same form (overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass). The creation of a new subclass which has some of the functionality (with selective modification) of another class allows software developers to easily customize existing code to meet their particular needs.

Although object-oriented programming offers significant improvements over other programming concepts, program development still requires significant outlays of time and effort, especially if no pre-existing software programs are available for modification. Consequently, a prior art approach has been to provide a program developer with a set of pre-defined, interconnected classes which create a set of objects and additional miscellaneous routines that are all directed to performing commonly-encountered tasks in a particular environment. Such pre-defined classes and libraries are typically called "frameworks" and essentially provide a pre-fabricated structure for a working application.

For example, a framework for a user interface might provide a set of pre-defined graphic interface objects which create windows, scroll bars, menus, etc. and provide the support and "default" behavior for these graphic interface objects. Since frameworks are based on object-oriented techniques, the pre-defined classes can be used as base classes and the built-in default behavior can be inherited by developer-defined subclasses and either modified or overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending the capabilities of the original program. In addition, developers are not blindly working through layers of code because the framework provides architectural guidance and modeling and, at the same time, frees the developers to supply specific actions unique to the problem domain.

There are many kinds of frameworks available, depending on the level of the system involved and the kind of problem to be solved. The types of frameworks range from high-level application frameworks that assist in developing a user interface, to lower-level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application frameworks include MacApp (Apple), Bedrock

(Symantec), OWL (Borland), NeXT Step App Kit (NeXT), and Smalltalk-80 MVC (ParcPlace).

While the framework approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over other programming techniques, there are difficulties which arise. Application frameworks generally consist of one or more object "layers" on top of a monolithic operating system and even with the flexibility of the object layer, it is still often necessary to directly interact with the underlying operating system by means of awkward procedural calls.

In the same way that an application framework provides the developer with prefab functionality for an application program, a system framework, such as that included in a preferred embodiment, can provide a prefab functionality for system level services which developers can modify or override to create customized solutions, thereby avoiding the awkward procedural calls necessary with the prior art application frameworks programs. For example, consider a display framework which could provide the foundation for creating, deleting and manipulating windows to display information generated by an application program. An application software developer who needed these capabilities would ordinarily have to write specific routines to provide them. To do this with a framework, the developer only needs to supply the characteristics and behavior of the finished display, while the framework provides the actual routines which perform the tasks.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system, including the application and the operating system. For the commercial or corporate developer, systems integrator, or OEM, this means all of the advantages that have been illustrated for a framework such as MacApp can be leveraged not only at the application level for such things as text and user interfaces, but also at the system level, for services such as printing, graphics, multi-media, file systems, I/O, testing, etc.

A preferred embodiment is written using JAVA, C, and the C++ language and utilizes object oriented programming methodology. Object oriented programming (OOP) has become increasingly used to develop complex applications. As OOP moves toward the mainstream of software design and development, various software solutions require adaptation to make use of the benefits of OOP. A need exists for these principles of OOP to be applied to a messaging interface of an electronic messaging system such that a set of OOP classes and objects for the messaging interface can be provided.

OOP is a process of developing computer software using objects, including the steps of analyzing the problem, designing the system, and constructing the program. An object is a software package that contains both data and a collection of related structure, and procedures. Since it contains both data and a collection of structures and procedures, it can be visualized as a self-sufficient component that does not require other additional structures, procedures or data to perform its specific task. OOP, therefore, views a computer program as a collection of largely autonomous components, called objects, each of which is responsible for a specific task. This concept of packaging data, structures, and procedures together in one component or module is called encapsulation.

In general, OOP components are reusable software modules which present an interface that conforms to an object model and which are accessed at run-time through a component integration architecture. A component integration architecture is a set of architecture mechanisms which allow software modules in different process spaces to utilize each others' capabilities or functions. This is generally done by assuming a common component object model on which to build the architecture.

It is worthwhile to differentiate between an object and a class of objects at this point. An object is a single instance of the class of objects, which is often just called a class. A class of objects can be viewed as a blueprint, from which many objects can be formed.

OOP allows the programmer to create an object that is a part of another object. For example, the object representing a piston engine is said to have a composition-relationship with the object representing a piston. In reality, a piston engine comprises a piston, valves and many other components; the fact that a piston is an element of a piston engine can be logically and semantically represented in OOP by two objects.

OOP also allows creation of an object that "depends from" another object. If there are two objects, one representing a piston engine and the other representing a piston engine wherein the piston is made of ceramic, then the relationship between the two objects is not that of composition. A ceramic piston engine does not make up a piston engine. Rather it is merely one kind of piston engine that has one more limitation than the piston engine; its piston is made of ceramic. In this case, the object representing the ceramic piston engine is called a derived object, and it inherits all of the aspects of the object representing the piston engine and adds further limitation or detail to it. The object representing the ceramic piston engine "depends from" the object representing the piston engine. The relationship between these objects is called inheritance.

When the object or class representing the ceramic piston engine inherits all of the aspects of the objects representing the piston engine, it inherits the thermal characteristics of a standard piston defined in the piston engine class. However, the ceramic piston engine object overrides these ceramic specific thermal characteristics, which are typically different from those associated with a metal piston. It skips over the original and uses new functions related to ceramic pistons. Different kinds of piston engines have different characteristics, but may have the same underlying functions

associated with it (e.g., how many pistons in the engine, ignition sequences, lubrication, etc.). To access each of these functions in any piston engine object, a programmer would call the same functions with the same names, but each type of piston engine may have different/overriding implementations of functions behind the same name. This ability to hide different implementations of a function behind the same name is called polymorphism and it greatly simplifies communication among objects.

With the concepts of composition-relationship, encapsulation, inheritance and polymorphism, an object can represent just about anything in the real world. In fact, our logical perception of the reality is the only limit on determining the kinds of things that can become objects in object-oriented software. Some typical categories are as follows:

- 10 — Objects can represent physical objects, such as automobiles in a traffic-flow simulation, electrical components in a circuit-design program, countries in an economics model, or aircraft in an air-traffic-control system.
- Objects can represent elements of the computer-user environment such as windows, menus or graphics objects.
- An object can represent an inventory, such as a personnel file or a table of the latitudes and longitudes of cities.
- An object can represent user-defined data types such as time, angles, and complex numbers, or points on the plane.

With this enormous capability of an object to represent just about any logically separable matters, OOP allows the software developer to design and implement a computer program that is a model of some aspects of reality, whether that reality is a physical entity, a process, a system, or a composition of matter. Since the object can represent anything, 20 the software developer can create an object which can be used as a component in a larger software project in the future.

If 90% of a new OOP software program consists of proven, existing components made from preexisting reusable objects, then only the remaining 10% of the new software project has to be written and tested from scratch. Since 90% already came from an inventory of extensively tested reusable objects, the potential domain from which an error could 25 originate is 10% of the program. As a result, OOP enables software developers to build objects out of other, previously built, objects.

This process closely resembles complex machinery being built out of assemblies and sub-assemblies. OOP technology, therefore, makes software engineering more like hardware engineering in that software is built from existing components, which are available to the developer as objects. All this adds up to an improved quality of the software as 30 well as an increased speed of its development.

Programming languages are beginning to fully support the OOP principles, such as encapsulation, inheritance, polymorphism, and composition-relationship. With the advent of the C++ language, many commercial software developers have embraced OOP. C++ is an OOP language that offers a fast, machine-executable code. Furthermore, C++ is suitable for both commercial-application and systems-programming projects. For now, C++ appears to be the most popular 35 choice among many OOP programmers, but there is a host of other OOP languages, such as Smalltalk, common lisp object system (CLOS), and Eiffel. Additionally, OOP capabilities are being added to more traditional popular computer programming languages such as Pascal.

The benefits of object classes can be summarized, as follows:

- 40 — Objects and their corresponding classes break down complex programming problems into many smaller, simpler problems.
- *Encapsulation* enforces data abstraction through the organization of data into small, independent objects that can communicate with each other. Encapsulation protects the data in an object from accidental damage, but allows other objects to interact with that data by calling the object's member functions and structures.
- 45 — *Subclassing* and inheritance make it possible to extend and modify objects through deriving new kinds of objects from the standard classes available in the system. Thus, new capabilities are created without having to start from scratch.
- *Polymorphism* and multiple inheritance make it possible for different programmers to mix and match characteristics of many different classes and create specialized objects that can still work with related objects in predictable ways.
- 50 — *Class hierarchies* and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.
- *Libraries* of reusable classes are useful in many situations, but they also have some limitations. For example:
- *Complexity*. In a complex system, the class hierarchies for related classes can become extremely confusing, with many dozens or even hundreds of classes.
- 55 — *Flow of control*. A program written with the aid of class libraries is still responsible for the flow of control (i.e., it must control the interactions among all the objects created from a particular library). The programmer has to decide which functions to call at what times for which kinds of objects.
- *Duplication of effort*. Although class libraries allow programmers to use and reuse many small pieces of code, each

programmer puts those pieces together in a different way. Two different programmers can use the same set of class libraries to write two programs that do exactly the same thing but whose internal structure (i.e., design) may be quite different, depending on hundreds of small decisions each programmer makes along the way. Inevitably, similar pieces of code end up doing similar things in slightly different ways and do not work as well together as they should.

5 Class libraries are very flexible. As programs grow more complex, more programmers are forced to reinvent basic solutions to basic problems over and over again. A relatively new extension of the class library concept is to have a framework of class libraries. This framework is more complex and consists of significant collections of collaborating 10 classes that capture both the small scale patterns and major mechanisms that implement the common requirements and design in a specific application domain. They were first developed to free application programmers from the chores involved in displaying menus, windows, dialog boxes, and other standard user interface elements for personal computers.

15 Frameworks also represent a change in the way programmers think about the interaction between the code they write and code written by others. In the early days of procedural programming, the programmer called libraries provided by the operating system to perform certain tasks, but basically the program executed down the page from start to finish, and the programmer was solely responsible for the flow of control. This was appropriate for printing out paychecks, calculating a mathematical table, or solving other problems with a program that executed in just one way.

20 The development of graphical user interfaces began to turn this procedural programming arrangement inside out.

25 These interfaces allow the user, rather than program logic, to drive the program and decide when certain actions should be performed. Today, most personal computer software accomplishes this by means of an event loop which monitors the mouse, keyboard, and other sources of external events and calls the appropriate parts of the programmer's code according to actions that the user performs. The programmer no longer determines the order in which events occur. Instead, a program is divided into separate pieces that are called at unpredictable times and in an unpredictable order. By relinquishing control in this way to users, the developer creates a program that is much easier to use. Nevertheless, individual pieces of the program written by the developer still call libraries provided by the operating system to accomplish certain tasks, and the programmer must still determine the flow of control within each piece after it's called by the event loop. Application code still "sits on top of" the system.

30 Even event loop programs require programmers to write a lot of code that should not need to be written separately for every application. The concept of an application framework carries the event loop concept further. Instead of dealing with all the nuts and bolts of constructing basic menus, windows, and dialog boxes and then making these things all work together, programmers using application frameworks start with working application code and basic user interface elements in place. Subsequently, they build from there by replacing some of the generic capabilities of the framework with the specific capabilities of the intended application.

35 Application frameworks reduce the total amount of code that a programmer has to write from scratch. However, because the framework is really a generic application that displays windows, supports copy and paste, and so on, the programmer can also relinquish control to a greater degree than event loop programs permit. The framework code takes care of almost all event handling and flow of control, and the programmer's code is called only when the framework needs it (e.g., to create or manipulate a proprietary data structure).

40 A programmer writing a framework program not only relinquishes control to the user (as is also true for event loop programs), but also relinquishes the detailed flow of control within the program to the framework. This approach allows the creation of more complex systems that work together in interesting ways, as opposed to isolated programs, having custom code, being created over and over again for similar problems.

45 Thus, as is explained above, a framework basically is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior (e.g., for menus and windows), and programmers use it by inheriting some of that default behavior and overriding other behavior so that the framework calls application code at the appropriate times.

There are three main differences between frameworks and class libraries:

50 *Behavior versus protocol.* Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides.

55 *Call versus override.* With a class library, the code the programmer instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the frame-

work rather than specifying how the different pieces should work together.

Implementation versus design. With class libraries, programmers reuse only implementations, whereas with frameworks, They reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

Thus, through the development of frameworks for solutions to various problems and programming tasks, significant reductions in the design and development effort for software can be achieved. A preferred embodiment of the invention utilizes HyperText Markup Language (HTML) to implement documents on the Internet together with a general-purpose secure communication protocol for a transport medium between the client and the merchant. HTTP or other protocols could be readily substituted for HTML without undue experimentation. Information on these products is available in T. Berners-Lee, D. Connolly, "RFC 1866: Hypertext Markup Language - 2.0" (Nov. 1995); and R. Fielding, H. Frystyk, T. Berners-Lee, J. Gettys and J.C. Mogul, "Hypertext Transfer Protocol --HTTP/1.1: HTTP Working Group Internet Draft" (May 2, 1996). HTML is a simple data format used to create hypertext documents that are portable from one platform to another. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of domains. HTML has been in use by the World-Wide Web global information initiative since 1990. HTML is an application of ISO Standard 8879:1986 Information Processing Text and Office Systems; Standard Generalized Markup Language (SGML).

To date, Web development tools have been limited in their ability to create dynamic Web applications which span from client to server and interoperate with existing computing resources. Until recently, HTML has been the dominant technology used in development of Web-based solutions. However, HTML has proven to be inadequate in the following areas:

- 25 o Poor performance;
- o Restricted user interface capabilities;
- o Can only produce static Web pages;
- o Lack of interoperability with existing applications and data; and
- o Inability to scale.

30 Sun Microsystem's Java language solves many of the client-side problems by:

- o Improving performance on the client side;
- o Enabling the creation of dynamic, real-time Web applications; and
- 35 o Providing the ability to create a wide variety of user interface components.

Java is compiled into bytecodes in an intermediate form instead of machine code (like C, C++, Fortran, etc.). The bytecodes execute on any machine with a bytecode interpreter. Thus, Java applets can run on a variety of client machines, and the bytecodes are compact and designed to transmit efficiently over a network which enhances a preferred embodiment with universal clients and server-centric policies.

40 With Java, developers can create robust User Interface (UI) components. Custom "widgets" (e.g. real-time stock tickers, animated icons, etc.) can be created, and client-side performance is improved. Unlike HTML, Java supports the notion of client-side validation, offloading appropriate processing onto the client for improved performance. Dynamic, real-time Web pages can be created. Using the above-mentioned custom UI components, dynamic Web pages can 45 also be created.

45 Sun's Java language has emerged as an industry-recognized language for "programming the Internet." Sun defines Java as: "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic, buzzword-compliant, general-purpose programming language. Java supports programming for the Internet in the form of platform-independent Java applets." Java applets are small, specialized 50 applications that comply with Sun's Java Application Programming Interface (API) allowing developers to add "interactive content" to Web documents (e.g. simple animations, page adornments, basic games, etc.). Applets execute within a Java-compatible browser (e.g. Netscape Navigator) by copying code from the server to client. From a language standpoint, Java's core feature set is based on C++. Sun's Java literature states that Java is basically "C++, with extensions from Objective C for more dynamic method resolution".

55 Another technology that provides similar function to JAVA is provided by Microsoft and ActiveX Technologies, to give developers and Web designers wherewithal to build dynamic content for the Internet and personal computers. ActiveX includes tools for developing animation, 3-D virtual reality, video and other multimedia content. The tools use Internet standards, work on multiple platforms, and are being supported by over 100 companies. The group's building

blocks are called ActiveX Controls, small, fast components that enable developers to embed parts of software in hyper-text markup language (HTML) pages. ActiveX Controls work with a variety of programming languages including Microsoft Visual C++, Borland Delphi, Microsoft Visual Basic programming system and, in the future, Microsoft's development tool for Java, code named "Jakarta." ActiveX Technologies also includes ActiveX Server Framework, allowing developers to create server applications. One of ordinary skill in the art readily recognizes that ActiveX could be substituted for JAVA without undue experimentation to practice the invention.

A preferred embodiment provides a system for building manageable applications. The applications can be readily deployed, on to a variety of platforms, and configured so that it's easy to partition them on to clients versus servers and administer the applications. A preferred embodiment is enabled as a client server application that's distributed nodes in a multi-node platform. A single application is divided up into pieces and distributed across nodes in the network. Thus, an application is defined as a distributed system.

Enterprise computing systems often a heterogeneous collection of nodes interconnected by a network. In a typical environment, a server node in the network holds data and programs that interact with the database, and a client node contains a program or programs for accessing the information on the server. The complexity of these environments makes it difficult to create, configure, deploy and administer software applications. However, the advent of Web technologies (browsers, the Java language and HTTP) has enabled enterprises to create and use internal Webs to assist in solving some of these problems. Java enables the Web as a client-server application platform and distribution channel to interact with a preferred embodiment in addressing many of the aforementioned challenges.

A preferred embodiment includes a toolkit for creating client programs that can be downloaded from the Web; interfaces for enabling a server program to function with a client as a single application; tools for connecting both client programs and server programs with a framework for executing the tools; and tools for installing, deploying and administering applications.

An application created in accordance with a preferred embodiment consists of a set of components that cooperate with each other. A server component can be implemented in any source language that can call a C program. A client component is implemented in the Java programming language. The client component consists of a GUI and a Presentation Engine (PE). To complete the application system, a preferred embodiment provides a communication layer that enables the exchange of messages between the client and the server components, an Exception layer for reporting errors, and an Access layer for managing application deployment. The task of an application developer utilizing a preferred embodiment is to assemble the components into an application system.

A preferred embodiment provides Java client access to server applications and can be utilized to create new applications or extend existing applications. Among existing applications, those that are partitioned so that server programs and databases are on the server, and user interfaces are on the client are especially suited for migration to the preferred embodiment. Other applications especially suited to a preferred embodiment require access to departmental or corporate data without changes to databases or the programs that access them. For enterprises already using the web technologies in an Internet, Intranet or other network environment, applications in accordance with a preferred embodiment provide quick access to existing data from anywhere in the organization or the world. Applications in accordance with a preferred embodiment can execute on any processor with the Java interpreter/runtime (Java JDK) or JavaOS installed. Application client programs are developed in Java using a null application template that contains the necessary Java classes and methods for integration with a Graphical User Interface (GUI). The template includes Java classes which will allow the client program to communicate with the server program. Scripts and tools for installing and deploying applications, include a generalized startup applet for application launch from a Web browser or applet viewer.

The primary development task in accordance with a preferred embodiment is to create an application front end referred to as a Presentation Engine, hereinafter (PE), from a provided template. The PE template includes methods (logic in an object) to send messages and their data through a client communication library. Developers modify the template to specify the messages and data required for their application. The communication library handles the passing of messages and data across the network. The PE toolkit supports these tasks. The toolkit is a full set of Java components which can be modified and extended for a particular application's requirements. The server program must also be enabled to communicate with the client by specifying which handler functions are called in response to inbound messages and inserting "send" calls to a server-local communication library when data are sent to a client. The two PE development tasks are to: (1) connect a Java UI to the PE framework, and (2) define/describe the messages to be exchanged between the PE and the server.

All of the components in accordance with a preferred embodiment reside on the server so that the server controls all access to its resources. The server thus controls deployment and administration of its set of applications. The clients use the web to access the server's resources. A template is also provided for creating an applet that enables users at client nodes to start applications from a browser which will be discussed in added detail below.

Figure 4 is a block diagram in accordance with a preferred embodiment. The client 410 and back end server 400 create and receive data messages communicated via messages at their local communication libraries 420. Messages are events which are encoded in a protocol, "ictp" layered on top of TCP/IP. The execution framework 430 layer could

be implemented on any network protocol as asynchronous, event-driven message passing on top of a communication protocol, such as TCP/IP, avoiding the dependencies of any particular server protocol. An application has a specific set of message events, and each component (front 410 and back 400 ends) includes local handlers for the set of message events. Thus, any two components can be plugged into the execution framework 430 to form an application 440 if they include local handlers for messages in the set of message events defined for the application 440. The components each include a local instance of a communication library 420. A component only interacts with the Application Programming Interface (API) of its local communication library 420 in order to send or receive message events.

Application components in accordance with a preferred embodiment are "loosely coupled" because there is no direct dependence (or communication) between the components. The execution framework 430 provides a consistent connection for all message transfer for the various applications. The execution framework 430 supplies additional services, such as error reporting, for the connection. An application 440 is insulated from the physics properties of the specific distributed platform on which the application is deployed. Thus, the application components and set of message events remain constant while the distributed platform can be scaled and modified.

Since applications 440 are systems containing well defined components as is shown in Figures 4 and 5, it is possible to install sets of front ends 530 and back ends 540 plus the definitions of how to associate them into applications 440. The server, therefore, holds all the resources while the client 500 can only request access to the resources. The server 520 is extended to allow it to control access to its resources by enforcing well defined policies.

Figure 5 illustrates how a preferred embodiment leverages Java to facilitate the establishment and implementation of server-centric policies. The client and server nodes communicate utilizing the web technologies in an Internet, Intranet or other network environment. The client node 500 contacts the server node 520 via HTTP with a request to execute an application. After authenticating the client 500, the server node 520 selects front 502 and back end 510 components based on the application definition list maintained at the server node 520. The server starts its back end process 510 and sends the front end program 502 to the client node via the Web technologies in an Internet, Intranet or other network environment. The client 500 executes the selected front end 502 locally at the client node 500. The front end (client) programs open a TCP/IP connection back to the server to initiate message passing in order to run the applications. The front end program 502 is implemented entirely in Java which facilitates instances of client/server applications which can run concurrently on a set of multi-platform clients. The server 520 is able to send a front end program 502 to any client node 500 which has the Java runtime installed on the computer. Server policies will not involve the clients. The policies will focus on the server's control of its local resources.

Figure 5 also illustrates the loose coupling between the front end 502 and the back end 510 application components. Each program includes a local communication library 515. The front end library is implemented in Java, the back end library is implemented in C++ with a C API. The programs each utilize their local communication library API to send and receive messages. Thus, the programs do not communicate directly with each other.

35 Presentation Engine

There are two phases for each application execution. In the first phase, application startup, the client node requests access to the server node's resources and the server acts on this request. The nodes are associated via the Internet or other communication network, and thus do not have a permanent relationship within the enterprise. In the second phase, application execution, the client has received a front end Java program called a Presentation Engine (PE) to facilitate presentation services. The front end and back end are communicating via the execution framework over TCP/IP.

Figure 6 illustrates the processing associated with application startup in accordance with a preferred embodiment. When an application is started, the client node executes a startup applet 620 which first collects information about the client 600 user and contacts the server node 610 via http 602. The server node 610 has been extended to include a web server 630 for processing requests via HTTP over the Web technologies in an Internet, Intranet or other network environment. The access layer 640 is called via a cgi-bin interface from the web server 630. The access layer 640 provides a framework so that the information about the client user, for example userid and password, can be used to call server-resident authentication services. Should authentication be successful, the access layer 640 uses the application name, which is also supplied by the startup applet, and invokes the app manager 650. The app manager 650 handles the application definitions installed on the server node 610. The app manager 650 selects the application back end 660 and initiates the processing. A listener socket is opened on the server node, and the port number is returned to the app manager 650. The app manager 650 also selects the application Front End (FE) 670, stored on the server as a set of Java bytecodes, and creates a PE 670 instance which includes the listener socket port number. Application startup ends when the PE 670 instance is downloaded to the client node for execution.

Application Execution

When application execution is initiated, the client node begins to interpret the PE 700 (Figure 7) it has received from the server node 710. The PE 700 is a framework (which includes an User Interface (UI) which can include a graphical UI (GUI)) and an instance of a communication library 720 implemented in Java. Once it starts up, the PE 700 opens a socket connection to the server node 710 utilizing the server port number it was supplied when the server app manager 650 started the back end process 730. The back end process 730 on the server node 710 may be a distributed system which encapsulates an interface to a Data Base Management System (DBMS) 712. The front end 700 focuses solely on presentation services and its own web access capabilities. The execution framework 724 specifically links the front end component 700 to the back end component 730 using event-driven message passing 735. This design preserves modularity between the front end 700 and the back end 730 of an application.

During application execution, the server node 710 communication library 720 manages the connection to facilitate policies that maximize access to its resources. For example, a server can be configured for a maximum time to await a transaction response. A timer runs, if it exceeds the maximum time before new client messages are received the server 15 will terminate the client connection and recycle the port. The execution framework 724 supplies runtime error reporting services and application execution data which are accessible to system administrators via Web technologies in an Internet, Intranet or other network environment access. When application execution terminates, the relationship between the client node 700 and the server node 710 also terminates. Application startup must occur each time application execution is desired.

A client's presentation engine can be stored on the client node and started dynamically if the presentation engine 20 is cached on the client. The reason for storing the presentation engine in cache is to support mobile (nomadic) computing, performance and to reduce network traffic. This technique would require versioning to assure that the presentation engine is synched with the latest, most current release on the server.

Application Development

The application development process in accordance with a preferred embodiment is a system integration task rather than a third generation language programming task. Figure 7 illustrates the three fundamental components of an application in accordance with a preferred embodiment. The front end, client side 700, the back end 730 server side 30 710, and the execution framework 724 which facilitates the connection between the front end and the back end. The application development process consists of a plurality of steps. The first step defines the responsibilities of the front and back end components. A preferred embodiment is designed to facilitate migration of existing client/server applications to function utilizing the Web technologies in an Internet, Intranet or other network environment for communication 35 services. This migration does not require redesigning existing application logic. The process entails the implementation of a new PE front end implemented in the Java language which exchanges message events with the remaining server back end code. Figure 8 illustrates the migration of an existing client-server application to one supported by a preferred embodiment.

The new Java PE front end 820 is added to the existing application code and will contain the Presentation Layer (UI tier) 800 and some amount of non-database aware logic responsible for manipulating variables. All logic which is 40 aware of the database and its data remains in the back end component. The PE 820 is not limited to logic from the existing application. It is a program implemented in Java, and thus can take direct advantage of additional Internet related services within the enterprise intranet. The complexity of the PE is determined by the developer's choice.

The second step in migrating existing applications to utilize a preferred embodiment is to define the message 45 events exchanged between the PE and the back end. The PE and the back end are loosely coupled and exchange data events. Each event has a unique name and data associated with the event. Message event data items are declared by type, all primitives plus one composite type. Message data types directly map to primitive types in both Java and C. The communication libraries have APIs which can be used by developers to define message events.

The third step is to set up the back end to handle message events. The server resident back end component must 50 handle message events. This processing is enabled by linking in an instance of the communication library with existing server code. The communication library has a C API, so the server code can be implemented in any language which can call C. The existing server code utilizes the communication library API to send and receive messages. The communication library calls handler functions in the server code when message events come in from the front end. The developer will create these handler functions and register them, one handler function per defined message event, using APIs in the communication library.

Once compiled and linked, the back end is a single process which is started by the app manager during application 55 initiation. The interface between the server communication library and the server code is multi-threaded.

The fourth step is to develop a Presentation Engine front end. Java is utilized for this task. Templates and tools are provided to facilitate this task. The detailed tasks associated with facilitating these tasks are described below.

The final step in migrating an existing client server application to a preferred embodiment is to install application components on the server node. Both the front end (client) and back-end components are installed on the server node. A Java startup applet enables clients to access the application on the server. A set of tools for application installation and configuration are provided as applications on server nodes in accordance with a preferred embodiment. A Java startup applet template is also provided to facilitate development of the application's startup applet.

5 Presentation Engine Development

10 Presentation engine development is the step of developing a new application front-end in Java. To simplify this task and provide application performance and robustness guarantees, all application front ends are instances of a single class of programs. A basic presentation engine framework implemented in Java is provided as an example in accordance with a preferred embodiment. Developing a specific application presentation engine means extending and customizing the basic presentation engine framework template. A Presentation Engine (PE) is itself a system with two components: (1) a UI implemented in Java, and (2) the PE framework. Developing a PE is a system integration task with 15 the following two steps.

(1) Develop a UI which is implemented with Java.

20 Figure 9 is a block diagram illustrating a PE 960 in accordance with a preferred embodiment. The UI 900 component is developed according to a developer's particular requirements. Since the front end processing utilizes the Java language, the UI also takes advantage of Java's rich functionality. The UI 900 is connected to the PE 960 framework via an interface to the UI Adaptor 910 component. The interface is a general message-passing interface that facilitates the attachment of any Java implemented UI 900 with the PE 960 framework to exchange data events. An API is provided to the PE 960 framework so that a developer can link the UI 900 to the UI Adaptor 910.

25 (2) Extend the PE Framework Template (960)

30 Figure 9 illustrates how the PE Framework 960 architecture is arranged to facilitate two kinds of external events in accordance with a preferred embodiment. The two kinds of external events are UI events and Message events from the Execution Framework. The framework includes a Java implemented communication library component, the Comm Adaptor 950. The PE framework 960 interfaces to the UI 900 via the UI Adaptor 910 and interfaces to the server TCP/IP connection via the Comm Adaptor 950. The mediator 920 component handles all external events moving into and out of the PE framework 960.

35 The UI 900, UI Adaptor 910 and the Model 940 are directly extended by the developer supplying application specific information. The communication library and mediator components are not extended by developers. The PE framework 960 has a model 940 -view 930 -controller architecture. A developer may extend the Model 940 component in order to implement local logic and maintain PE-local state. The view component 930 maps between the Model 940 data representation and the message data representations.

40 It is possible to create a PE framework without extending the Model 940 component of the PE Framework 960. This model is referred to as a "PE-lite" option and consists of the comm adaptor 950, mediator 920, UI Adaptor 910 and the UI 900. To utilize this option, a developer assembles the components from a UI 900 and a PE framework instance 960. The PE framework 960 makes use of the Mediator 920 to directly map between UI external events and Comm external events.

45 The execution framework 960 is a communication abstraction that utilizes TCP/IP as a communication backbone, and is analogous to a bus in a computer system. Each component in accordance with a preferred embodiment plugs into the execution framework much as various function cards plug into the bus of a computer system. There is no notion of local versus remote in the components which is unlike a typical distributed computing environment.

50 In this architecture, there is no distributed domain. Everything is local and the way that a component connects up with this framework is by making local calls to a communication layer which is the framework. It's all embodied in the communication library components. So, the messages that are exchanged between programs are data. Rather than function calls, they are actually data containers. All that can happen in the exchange is that data are packed up and sent as an event, or data are packed up here and sent over there as an event.

55 Inside of the client and server programs, there are handlers for the data in the message events. So, any front end and any back end has local handlers for a common set of message events. They can be plugged together by the framework to create an application. So, what we call these is, "loosely coupled", which means that there are no dependencies between the programs and they do not communicate directly. This preserves modularity in the application. A program may send data events, and the program also is enabled to handle data events which interrupt its processing.

The framework currently is layered over TCP/IP for a communication vehicle, but other networking protocols can be

utilized. The programs always interact with the ICE-T Comm Layer. There is a firewall between the actual, physical distributed platform, underneath and the application above because the application components are interfacing with the abstraction. Thus, applications are independent from the details of the physical structure of the distributed enterprise. The enterprise may be scaled, and/or modified without requiring redesign of the applications.

5

Example In Accordance With A Preferred Embodiment

Figure 10 is a block diagram of a prior art client server architecture. In the prior art a client computer 1000 would access a network 1030 via a predefined data protocol (datastream) and interact with a predefined server 1001 containing code 1010 and data in the form of a DataBase Management System (DBMS) 1020.

An application in accordance with an alternate embodiment is shown in Figure 11. The encapsulated DBMS 1102 is conserved, as is the enormous investment in code 1170. However, the server 1180 includes a service API 1160 for communicating to the network support 1150 which is responsible for turning the web node into a loosely coupled client 1120 utilizing a small amount of Java code and conforming to a predefined datastream that is sent by the server 1180 to the client 1120. A web server 1106 is utilized to facilitate access to and transmission of messages over the web from the server 1180 via the access module 1108 and the PE framework 1104 as discussed earlier and reiterated below.

Figure 12 illustrates a server 1210 establishing contact with a client 1200 in accordance with an alternate embodiment. The code 1240 and DBMS 1230 are as described in Figures 10 and 11. However, web contact is established between the nodes by authenticating a user utilizing the Access Layer 1280 to establish client 1200 - server 1210 communication. Then, the Java bytecodes 1250 are extracted from the PE Repository 1290 and downloaded to the client 1200 using the Access Layer 1280 and the Web Server 1270.

Figure 13 illustrates a loosely coupled client - server application in accordance with an alternate embodiment. The Server 1300 communicates via a service API 1360 and Network Support 1340 to a Client 1330 utilizing a predefined data stream 1350 and application code 1320 and a DBMS 1310. The "loosely" coupled nature of the application architecture is enabled through message communication in an asynchronous manner letting the Client 1330 and the Server 1300 maintain state independence and connect via a low bandwidth network 1350. The applications require no function call level API's.

Some of the basic architectural design decisions that form the foundation of a preferred embodiment include the Server 1300 controlling both the Client 1330 and Server 1300 states. The Java language is utilized to communicate via the Web technologies in an Internet, Intranet or other network environment to distribute Client 1330 state information to the Client node. The Server 1300 presents an encapsulated DBMS interface to the network. The Server 1300 node is extended by a framework to support the establishment of a Client 1330 - Server 1300 relationship for web nodes.

The Client 1330 - Server 1300 relationship for web nodes is established utilizing a secure http process which authenticates the user at the requesting node. Then, the Client 1330 node is established by selecting an appropriate Client 1330 state from stored states at the Server, and the Java client state is downloaded over the network to the particular Client 1330 node. Next, the Client 1330 - Server 1300 network communication session is commenced by starting the server process and establishing a socket connection to the Client 1330 node. Once the session is established, then the network must be managed by maintaining event-driven message passing with the PE communication library.

Figure 14 illustrates the system integration task necessary to develop an application 1430 in accordance with a preferred embodiment. First, the developer defines the message events 1450 that must be exchanged between components. The client program 1410 and server program 1420 are defined as applications 1430 on the Server Node 1420. The back end must be enabled to send and receive messages. The front end (PE) must be developed, then the components are installed in the server node 1420. The goal is to support modular application development and require minimal additional knowledge burden (e.g. sophistication in Java or OOP) from developers.

Figure 15 is a block diagram illustrating the modular design of a client application (PE) in accordance with a preferred embodiment. The application logic 1530 is written entirely in Java to maximize access to Internet facilities. The application logic runs as a Java program or an applet. An External Interface Library 1520 contains various callable functions for interfacing to the UI or API 1500 through the UI layer 1510. A Message Library 1540 provides messages for use in communicating information through the communication layer 1550 to the Server 1560. The architecture is a client loosely coupled with the server 1560. Data exchange is via event-driven message passing. A flexible UI 1510 and communication layer 1550 facilitates message passing APIs interfacing to a Server or UI designs. Developers are not required to understand network communication or system programming in order to design and implement applications. The client program can be designed to support efficient presentation services. The DBMS interface is encapsulated in the back end (server). Thus, there are no distributed transactions which must be handled in the web connection between client and server.

Figure 16 is a block diagram of a framework in accordance with an alternate embodiment. Model data 1600 is provided in the form of predefined classes that can be accessed and modified by a user to add additional data or logic to meet the requirements of a particular application and specify appropriate maps for views or UI components. All of the

class templates necessary to be extended into a fully functional model are provided. The Model data 1600 is utilized on the external IF View 1622 and additional model data 1620 may be added as necessary to expand the view. Similarly, the message view 1612 can be expanded by adding additional model data 1610 to the message view 1612. A mediator 1630 is responsible for routing messages and granting control to the Communication Adaptor 1640 or the UI Adaptor 1650.

Figure 17 illustrates the basic building blocks in accordance with an alternate embodiment. A PE object 1700 is an object containing methods (logic) and data (public and private information) that is utilized to facilitate a reusable, extensible function. A PE function 1710 utilizes PE objects 1700 to implement callbacks, observers, event handlers and maps in accordance with a preferred embodiment. PE Info 1720 is a basic unit of data moved between functional components, and a PE Event Handler 1730 is used by views to handle incoming messages and UI events in accordance with a preferred embodiment.

Figure 18 is a block diagram highlighting the steps utilized to extend the (PE) framework in accordance with a preferred embodiment. A user must modify the UI Adaptor 1800 as shown at 1810. All of the base processing necessary to facilitate the application is incorporated into the basic framework which provides all of the basic classes for instantiating objects necessary to perform user's requirements.

Figure 19 is an illustration of a PE Object 1900 in accordance with a preferred embodiment. The PE object is a data structure library that is used for building the model and other structured data passed around in messages. There is an associated containment hierarchy with primitive 1910 and composite 1920 objects instantiated from the base PE Object 1900. An object-oriented architecture facilitates polymorphic, encapsulated objects that are fully self-describing and fully enabling for a serialization interface.

Figure 20 is an illustration of a PE Event Handler 2000 used by Views to handle incoming messages and UI events in accordance with a preferred embodiment. The PE Event Handler 2000 dispatches tasks based on a name field and data defined in objects PEInfo 2010 data which is passed utilizing a message to implement a map to the model. Figure 21 illustrates a PEInfo object 2100 data in accordance with a preferred embodiment. The data encapsulates event and message signatures and is used in three places for passing between model and XIF View, Model and Msg View and XIF View and the UI.

Figure 22 illustrates incoming message flow to a model in accordance with a preferred embodiment. A message flows into a communication layer in bytestream format 2200 and passed via a mediator to a message view 2210 to unpack and dispatch the message utilizing a MsgHandler PE Event Handler to transfer the PEInfo 2220 data. The PEInfo 2220 is used to encapsulate the event and message signatures and transform the message into a model map 2230 which is thereafter utilized for setValue and getValue processing 2240. When a UI Event, such as a key press or other activity occurs at 2260, a callback PE Function 2270 utilizes PEInfo 2280 to pass event and data information to the external view dispatcher 2290 for transformation into information that the Model PEOBJect 2250 can utilize for further message processing.

Figure 23 illustrates incoming messages mapping a UI to a Model in accordance with a preferred embodiment in a manner very similar to Figure 22. Figure 24 illustrates outgoing messages mapping a model to messages in accordance with a preferred embodiment in a manner similar to Figures 22 and Figure 23. Figure 25 illustrates outgoing messages mapping a model to a UI in accordance with a preferred embodiment in a manner similar to the foregoing Figures. To further clarify processing in accordance with a preferred embodiment, the detailed installation and logic specification for an application in accordance with a preferred embodiment is presented below.

Enterprise computing environments usually consist of many kinds of nodes interconnected by a network. In a typical environment, a server node in the network holds data and programs that interact with the database, and a client node contains a program or programs for accessing the information on the server. The complexity of these environments makes it difficult to create, configure, deploy, and administer software applications. The advent of Web technologies (browsers, the Java language, HTTP) has enabled enterprises to create and use internal Webs to help solve some of these problems.

Java enables the Web as a client-server application platform and distribution channel. The "Enterprise" Computing Environment Toolkit (ICE-T) enables building, extending and deploying client-server applications for the Web. ("Enterprise" combines the internet and the enterprise.)

The ICE-T application provides:

- A toolkit for creating client programs that can be downloaded on the Web
- Interfaces for enabling a server program to work with a client as a single application
- The tools to connect both client program and server programs to a framework for executing them
- Tools for installing, deploying, and administering applications

ICE-T Applications

An application consists of a server program component (implemented in any language that can call C) and a client program component (implemented in Java). The client component consists of a GUI and a Presentation Engine (PE).

5 To complete the application system, ICE-T provides a Communication Layer that enables the exchange of messages between the client and server components, an Exception Layer for error reporting from ICE-T components, and an Access Layer for managing how applications are deployed. The task of the ICE-T application developer is to assemble the components into an application system.

ICE-T provides Java client access to server applications. New applications or existing ones extended utilizing a preferred embodiment. Among existing applications, those that are partitioned so that server programs (business logic) and databases are on the server, and user interfaces are on the client (three-tier client-server applications) are especially suited for migration to ICE-T front ends.

ICE-T is especially suited for enterprise applications that require:

15 • Access to department or corporate data without changes to databases or the programs that access them.
For enterprises already using an enterprise web technologies in an Internet, Intranet or other network environment, ICE-T applications can provide quick access to existing data from anywhere in the organization or the field where there is an internet connection.

20 • Client platform independence.
ICE-T Presentation Engines can run anywhere the Java Virtual Machine is present.

25 • Rapid development.
Clients are developed in Java using a null application template that contains the necessary Java classes and methods for integration with a GUI and a Communication Library.

30 • Easy deployment.
ICE-T provides scripts and tools for installing and deploying applications, include a generalized startup applet for providing application launch from a Web browser or applet viewer.

35 • Centralized authentication of users.
A customizable Access Layer, installed on the server, enables centralized control of access to client programs.

40 • Easy maintenance.
For most enterprises, maintaining existing applications is a tremendous resource burden. ICE-T provides the means to make new application front-ends (clients), or migrate existing ones, without changing the architecture or programs of the back-end (server), or requiring a steep learning curve.

45 • Wide use throughout the enterprise, from the desktop or the laptop. A familiar and pervasive interface.
End-users can name ICE-T applications as applets in a Web browser.

ICE-T Application Development and Deployment

The primary development task in an ICE-T application is to create an application front end, a Java Presentation Engine, from the provided template. The Presentation Engine (PE) template includes methods to send messages and their data through a client Communication Library. Developers modify the template to specify the messages and data required for their application. The Communication Library handles the passing of messages and data across the network. The Presentation Engine Toolkit supports these tasks. The Toolkit is a full set of Java components some of which you must modify or extend.

50 An additional development task is to modify the server program to specify which function to call in response to inbound messages and make calls to a server Communication Library to send results to the client. All of the components in an ICE-T application system reside on the server. To deploy an application, you install its components and additional ICE-T files and programs that manage applications on the server. ICE-T also provides a template for creating a startup applet that enables users to start applications from a browser. Chapter 3, "Configuring and Deploying ICE-T Applications" describes these tasks and tools.

Event-Driven Message Passing in ICE-T Applications

The components of ICE-T applications create, send, and receive messages in response to external events. An event is a way of communicating that something has happened, such as input from the user (a mouse click) or a change in the system environment (a server shutdown).

5 The ICE-T Communication Layer enables asynchronous event-driven message passing between client and server program components on TCP/IP. In ICE-T, the messages themselves are events, and each program component includes local handlers for message events.

Client and server program components must each be set up to create, send, and receive messages. Part of the

10 Presentation Engine's functionality is to determine the recipients and deliver the messages to them. For example, the user interface sends a message to the Presentation Engine when a user clicks a button. The Presentation Engine receives the message and then either sends it to the data model or the server program, which then performs an operation and replies with the result.

The server program must also be able to receive messages and must register functions to handle them.

15 The recipients of messages do not need to know much about each other, they just need to specify what messages they want to receive. This information is registered using the `createMessageHandlers()` method in the client program and the `createMessageHandlers()` function in the server program.

Presentation Engines should include handlers for two kinds of application events:

20 • Events from the user interface
Events coming in to the Presentation Engine from the user interface result in messages to the server or the Presentation Engine's data model.

25 • Events from the server
Events coming in to the Presentation Engine from the server result in displaying data in the user interface or putting data in the Presentation Engine's data model.

Server programs should include handlers for messages from the client. Typically, the handler would call application logic and send the resulting message and its data back to the client through the server Communication Library.

30 ICE-T Application Execution

ICE-T applications are designed to work within existing client-server environments. They differ from familiar client-server applications in some key ways.

35 Typically client programs are developed, maintained, and distributed periodically, taking long cycles of development time, and requiring time to deploy to client nodes. Users who are not on a node on the distribution route may miss software updates. Development can consume resources because of the difficulty of the language or tools used.

Compiled ICE-T Presentation Engines are installed on the server and downloaded on request through HTTP servers. A pair of Communication Libraries behave as a framework for executing the application. This communication layer 40 handles the marshaling and unmarshalling of the message data transferred between client and server. Both client and server should be prepared to handle shutdown events. ICE-T provides default shutdown handlers for this purpose, and developers can add their own.

45 How ICE-T Applications Work

Before developing an ICE-T application, you might find it useful to see how ICE-T applications work, from both an end-user's perspective and inside ICE-T.

50 The User View

ICE-T applications can use a Java-enabled Web browser for client access to application execution. Although developers may choose to have applications launched outside a browser, a Web page presents a familiar and easy to use interface for launching applications.

55 The user begins by opening a Web page and clicking on the URL for the application she wants to run. The URL is the address for a Web page that includes an ICE-T application startup applet. The Web page with the startup applet is loaded into the browser. The applet collects access information from the user. The applet contains the URL of the server holding the application components and the application name. This information is processed on the server. If the user name, password, and chosen application are authorized, the server downloads a Presentation Engine to the user's

node.

The Presentation Engine presents the user with an interface for interacting with the server program and data. It also determines where to pass messages for handling, either to the server program or to its own data model. One example of a client program is one that communicates with a server program that searches a database, for example an employee database. Such a client program might have these user interface elements:

- A text field where users enter a first or last name that they wish to find in the employee names database
- Buttons for clearing the fields (Clear), exiting the application (Quit), and launching the search (Search)
- A scrolling window for viewing the results of the query

The user enters a first or last name and presses Return or clicks a Search button. The client program sends the query to the server, where the server program searches the employee database for a matching name. If a match is found, the server returns the results to the client. The results are displayed in a window on the client.

The ICE-T View

When a user launches an ICE-T application, the client node establishes a Web connection with the server node using HTTP. The server manages this Web connection. ICE-T applications can be launched from a browser, an applet viewer, or as standalone applications. Figure 26 illustrates the steps associated with launching an application URL in accordance with a preferred embodiment. On the server side, the ICE-T Access Layer (a cgi-bin executable) authenticates the user data. If the authentication succeeds, the Access Layer contacts the ICE-T Application Manager and the Application Manager starts the server program and initiates a network session.

The Application Manager downloads an HTML page with a startup applet for the application. When the user runs the startup applet, the Application Manager selects a compiled Presentation Engine and downloads an HTML page containing the applet tag for it to the client using HTTP. The compiled Presentation Engine includes a GUI and an instance of the client Communication Library and is ready for execution in a Java-enabled browser or anywhere the Java Virtual Machine is installed.

The client node then executes the Presentation Engine locally. The Presentation Engine makes a TCP/IP connection to the server where the server program is running, and the client and server programs cooperate to execute the application.

When a user interface event occurs—for example, when the user enters a first name in the text field and clicks a Search button—the user interface passes a message to the Presentation Engine. The Presentation Engine either sends the message to its data model for processing by the client, or passes the message to the server for processing by the server program. The Presentation Engine determines where on the client a message is handled based on how you have registered message handlers. When the server program sends a message with data to the client, the Presentation Engine displays the result.

The exchange of messages between client and server is handled through the ICE-T Communication Libraries in the ICE-T Communication Layer. When the client program terminates, the Application Manager closes the socket connection to the client and terminates any server processes it started.

ICE-T Task Summary - Building Program Components

ICE-T's modular architecture makes it easy to distribute development tasks to multiple developers. Alternatively, a single developer can complete ICE-T development tasks in stages. Developing a client program requires making a Java Presentation Engine, connecting it to a user interface and to the ICE-T Communication Layer so that it can communicate with a server program. The server program could be an existing program or a new one. It must be written in a language that can call C so that it can work with the ICE-T Communication Layer. You don't need to develop new server programs for ICE-T applications, but you must enable the server program to handle messages from the client.

ICE-T Application Building Blocks

An ICE-T application consists of a client program and a server program communicating through a Communication Layer. The client program consists of:

- A GUI built with Java

- A Java Presentation Engine built using a template

These components, and related classes used by the Presentation Engine, combine to behave as a single client under the control of the Presentation Engine. The Presentation Engine may be presented as an applet launched from an applet viewer or a Web browser, or as a standalone application.

5 The server program, new or existing, is developed however the developer chooses. It must be in a language that calls C and it must include functions for handling messages from the client. A template with these functions is provided, as is a main routine that makes calls to the provided server Communication Library.

ICE-T provides these templates, tools, and libraries for developing applications:

10 • `pe_template.java`

A template for a working Presentation Engine.

- ICE-T packages (supplementary to the standard Java packages)

15 • `server-template.c` and `server_template.cc`

Server program templates (one each for C and C++) that define and enable message passing to and from the client. The templates can be used with existing programs or used as a starting point for developing server programs. These templates are analogous to the `pe_template` used for the client.

20 • ICE-T message data types that work the same on both client and server.

ICE-T also provides a framework that the developer does not modify and in which ICE-T applications can execute. This framework includes:

- Communication Layer

25 Supports network communication between client and server programs. The server Communication Library presents a C API to the server program, which is linked to this library. The client Communication Library is implemented in Java.

- ICE-T Exception Layer

30 Provides exception reporting from the ICE-T application (client and server) in addition to standard system error reporting.

The Presentation Engine

35 Every Presentation Engine extends (inherits from) a Java class named `PresentationEngine`. All of the objects that the client program needs are either in the `PresentationEngine` class or called by it. ICE-T provides a class that extends `java`, and calls the methods that you need to create a working Presentation Engine.

The filename for the Presentation Engine template is `pe_template.java`. You can find it in the ICE-T application installation directory under `Templates/C` or `Templates/C++`. The file is placed in each of the `Template` subdirectories for convenience. The `pe_template` is the same in both files.

40 Figure 27 describes the forms of a Presentation Engine, as an abstract Java class, a template for development, and an executable component in an application in accordance with a preferred embodiment.

Figure 27 describes the forms of a Presentation Engine, as an abstract Java class, a template for development, and an executable component in an application in accordance with a preferred embodiment.

To create a working Presentation Engine, you copy the `pe_template` file and make these changes:

45 • Supply your own Presentation Engine name.

- Create user interface components or map the ones from a Java GUI builder.

- Create a data model (if your application requires client-side processing).

50 • Define messages and their handlers.

- Register message handlers.

55 These steps are described in this chapter.

Presentation Engine Implementation Options

Developers can create Presentation Engines that simply send messages from the user interface to the server and display the data that the server returns on the client. This option for Presentation Engine implementation is called a "PELite." An alternative implementation of Presentation Engine handles some local data storage in what is called the Presentation Engine's data model. For the latter, developers are required to implement a `createModel()` method, described in this chapter. Both options are supported by the `pe_template`.

ICE-T Classes and Packages

The documentation for the ICE-T Presentation Engine API is presented in HTML form, like the Java API documentation, and is accessible from the following URL:
`file:///ICE-T Installation Directory/doc/api`
 where (ICE-T Installation Directory) is the name of the directory on your system where you have installed ICE-T. The C and C++ header files used by the server program are in the ICE-T installation directory under Server.

ICE-T Message Data Types

ICE-T message data types:

- Can be used to construct the data model in the Presentation Engine
- Are available for use in the server program for use in application logic
- Have analogs on both the client (Java) and server (C)
- Use basically the same APIs on both the client and server
- Can contain only other ICE-T data types
- Are used in messages from client to server, and server to client, where they will appear as a local data structure of their analogous type.

The table appearing below describes the primitive ICE-T message data types and their analogous types on client and server. The prefixes Pe and Srv are provided for your convenience in naming things. They are handled as the same type by ICE-T.

ICE-T Message Types (Primitive)			
	Data Type	Client (PE)	Server (C)
	PeBar, SrvChar	char (Unicode) 16 bits]	char (C) 8 bits
	PeString, SrvString	string	char*
	PeBoolean, SrvBoolean	Boolean 16 bits	int 1 byte
	PeInteger, SrvInteger	int 32 bits	int 32 bits
	PeLong, SrvLong	long 64 bits	long 64 bits
	PeFloat, SrvFloat	float 32 bits	float 32 bits

(continued)

ICE-T Message Types (Primitive)		
Data Type	Client (PE)	Server (C)
PeDouble, SrvDouble	double 64 bits	double 64 bits

1. ICE-T transmits only the ASCII (8 bits).

10 ICE-T supports a composite data type based on the Java vector class (java.util.Vector). It differs from the Java vector type in that it provides an implementation that works on both sides of the ICE-T application (Java clients and C or C++ server programs), and it only takes ICE-T primitive data types.

15

Table ICE-T Vector Type	
Data Type	Description
PeVector,	A collection that implements a variable-length array.
StyVector	Add elements sequentially with the add Element () method. Access an element by its index.

Working with the ICE-T Directory Structure

25 Before developing applications, copy the provided templates and Makefiles to an application development directory.

There are two subdirectories of templates.

- `(ICE-T Installation Directory)/Templates/C`
30 Contains `Example.mk`, `pe_template.java`, and `server_template.c`.
- `(ICE-T Installation Directory)/Templates/Cplusplus`
Contains `Example.mk`, `pe_template.java`, and `server_template.cc`.

35 For example, create an application directory for an application named `myAppName` in which the server program is written in C++, and copy the templates to it:

% `mkdir myAppName`

% `cd (ICE-T Installation Directory)/Templates/C++/ (ICE-T Installation Directory)/Applications/myAppName/`.

The ICE-T installation scripts and Make-hierarchy depend on the newly created application directory being two levels below the ICE-T installation directory.

40 If you choose not to follow the directory setup suggested by the example, you will have to supply options or arguments indicating where the ICE-T installation directory is in relationship to the directory you have created.

Designing the Client Program

45 Here are some design decisions to make before you develop the client program for an ICE-T application.

- Using a graphical user interface (GUI) toolkit, specify and design the user interface.
- Determine the events the client and server components of the application should handle. Name each of these events and associate data with it. In this step you answer the question of what messages will be passed between the Presentation Engine and the user interface and between the Presentation Engine and server.
- Decide what application logic to put in the client program, if any.
55 Decide what, if any, data processing you want the client program to handle. This affects whether you create event handling for data updates in the Presentation Engine or just use the Presentation Engine as a mediator between the user interface and the server data (a "PE Lite").

- Specify how the user interface should be updated when the Presentation Engine receives messages and data from the server.
- Specify how the Presentation Engine receives and decodes data from the server and whether it stores the data for later retrieval or displays the data directly.
- If you choose to store data in the Presentation Engine, specify how it is to be stored and updated and what other components in the application depend on

10 that data. **Developing the User Interface**

You can create the user interface for the ICE-T application at the same time as, or before, you create the Presentation Engine. Even though they can be separate, the Presentation Engine and user interface work together, so the interactions between them should be planned.

15 **Creating a Presentation Engine**

Create a Presentation Engine involves the following basic steps that are described in detail throughout this section:

1. If you have not done so already, copy one of the Templates subdirectories to the Applications directory.
The ICE-T installation directory includes a communication directory for developers to use. To create applications, make a subdirectory under applications for each ICE-T application you develop.
2. Modify and rename the pe_plate file to suit the application. Use the same name for the class and the file.
3. Create a Presentation Engine class.
Create your own Presentation Engine definition using pe_template.java.
4. Integrate the Presentation Engine with the user interface (GUI).
Create a separate user interface class using your chosen Java tool. Integrate the GUI with the Presentation Engine by implementing the createUI() method that is found in pe_template.java.
"Working with the ICE-T Directory Structure" describes how to implement createUI().
5. Determine and define the message events to be passed between the Presentation Engine and server program and the Presentation Engine and user interface.
Name each of these events and associate data with it. Specify the type of data that will be passed in response to each message event.
 - a. Add handlers for events on GUI components.
Implement the operations you want the application to perform in response to user interface events.
 - b. Write the code for handling incoming message events in the Presentation Engine.
Define handlers in the Presentation Engine for incoming messages from the GUI. Define handlers in the Presentation Engine for incoming messages from the server.
Developers can choose to write a separate class, or classes, in separate files to handle events. This means that a named handler can be modified without modifying the Presentation Engine framework itself.
6. Register which handlers will be sent which messages when events occur (map messages to handlers).
Specify what handler should receive each message.
"Registering Handlers" describes this step.
7. Build the Presentation Engine.
ICE-T provides a makefile to build both server and client programs.

55 These activities are analogous to the tasks for modifying server programs to work with clients. In both client and server program cases, ICE-T provides templates with methods (Presentation Engine) and functions (server) for registering handlers. Developers provide only the application-specific messages and data.

Creating a Presentation Engine Class

Copy one of the Templates subdirectories to the Applications subdirectory. There are two subdirectories, one for C and one for C++. For example, create a directory for an application named myAppName. If the server program for

5 myAppName is to be written in C, copy all of the files from the Templates/C directory:

```
% lnkair myAppName
% cp $(ICE-T Installation Directory)/Templates/C/*
$(ICE - T Installation Directory)/Applications/myAppName/.
```

For each Presentation Engine you create, modify pe_template.java to declare a class that extends the abstract

10 Java class PresentationEngine:

```
public class myPresentationEngine extends PresentationEngine
{ // methods for your application
}
```

Note - Be sure to give the class and the file the same name. For example, if the Presentation Engine class is named

15 myPresentationEngine, the file should be named myPresentationEngine.java.

pe_template.java contains the class and method declarations that you implement to provide the functionality you want.

The methods must include:

- 20 • createUI()
- createModel()
- createMessageHandlers()
- 25 • initializeApplication()

Implementing createModel() and initializeApplication() is optional.

You do not have to use initializeApplication() unless your client program requires local initialization before communication is started.

Importing Packages

The pe_template imports the appropriate packages, including these ICE-T packages and standard Java packages;

- 35 • sunsoft.ice.pe
- sunsoft.ice.pe
- 40 • java.net
- java.io
- java.applet
- 45 • java.util
- java.awt

50 The Java language package (java.lang) is implicitly used and does not have to be explicitly imported.

To include the required packages, a Presentation Engine class must have these import statements. Don't delete them from the pe_template.

```
import sunsoft.ice.pe.*;
import java.net.*;
import java.io.*;
```

```

import java.applet.*;
import java.util.*;
5 import java.awt.*;

```

Integrating the User Interface with the Presentation Engine

10 ICE-T applications must make user interface elements from the application GUI available as components that the Presentation Engine can access. The Presentation Engine reads and handles data from events in the user interface and updates the user interface with data returned from the server.

Creating a GUI that is separate from the Presentation Engine, offers developers more flexibility because updates to the GUI do not require major changes to the Presentation Engine. Use the `createUI()` method to make the GUI components available to the Presentation Engine.

15 To implement the `createUI()` method with application-specific operations, write a body for the method in which you:

- Declare variables for the user interface elements
- Create a frame to hold the GUI applet (optional)
20 - instantiate and initialize the GUI applet
- Get the user interface components
- Provide initial values for fields in the GUI (optional)
25 - Add the GUI elements to a Container object (provided in the `PresentationEngine` class)

30 For example, here is excerpted code from a `createUI()` method that declares variables for labels and text fields, creates a frame, and adds the user interface elements to a container component defined in the `PresentationEngine` class.

```

protected void createUI () {
35
    Label label_1, label_2, label_3;

```

40

45

50

55

```

TextField textField_1, textField_2, textField_3;

5      //... additional variables

fr = new Frame ("My Application Title");

10     fr.add ("Center", ui);

fr.pack (); fr.show();

15     //...

textField_1.setText (" ");
20     textField_2.setText (" ");

//...

25     uiContainer.addObject ("firstName", textField_1);
         uiContainer.addObject ("lastName", textField_2);

30     //...
         }

35

```

Creating a Data Model in the Presentation Engine (Optional)

40 Updates to the client program can be sent to the user interface or to the data model in the Presentation Engine. If your application is designed to hold data in the Presentation Engine, you use the `createModel ()` method to create the data objects (Observables) to hold the data and attach Observer objects to them.

`createModel ()` is optional and is commented out in the `pe_template`. To use `createModel ()`:

45 • Uncomment the method in the `Pe_template`

- Create Observable data objects to hold data from the server
- Attach Observers to the data objects, if necessary
- Put the Observable data objects into the data model

50 This example instantiates a `PeVector` data object named `empvector`. Using the `Observer` and `Observable` methods is described in "Attaching Observers to Data Objects."

55

```

protected void createModel () {

5           //create the observable
    PeVector empVector = new PeVector();

10          //Attach observer to the observable
    empVector.addObserver (new empObserver (model));

15          //put the observables into Model
    model.addObserver ("empVector", empVector);

```

20 Attaching Observers to Data Objects

Developers have the option of defining any data objects in the model as a simple PeObject (a data item that is sent in messages) or as a PeObservable, a data item with a method associated with it. A PeObservable is a data object with an associated PeObserver function that is called when the data is updated. If a data object changes state, and you want the application to notify any program components that depend on that data, you define each data object as a PeObservable and you attach a PeObserver to it.

ICE-T provides the peObservable and PeObserver classes. Developers call the methods in these classes to add observable data objects to the model and add their observers to an observer list.

30 Handling Events

ICE-T client programs receive messages in response to external events. Client program developers need to specify what messages they want the Presentation Engine to receive and register this information using the createMessageHandlers () method. Server program developers have the analogous task of defining message handlers and registering their handlers.

ICE-T provides a layer of abstraction to handle interactions between the GUI and the Presentation Engine. This layer is called the UI Adaptor and it is a set of Java classes that developers use as given. The Presentation Engine makes calls to the UI Adaptor to register messages. The Pe_template includes the necessary methods; developers provide the message bodies as described in the procedures in this chapter.

Presentation Engines handle two kinds of application events:

- Events from the user interface

Events coming in to the Presentation Engine from the user interface result in messages to the server or the Presentation Engine's data model.
- Events from the server

Events coming into the Presentation Engine from the server result in displaying data in the user interface or putting data in the Presentation Engine's data model.

50 Handling Events from the User Interface

Applications handle user interface events, such as button clicks and text entry. Java provides an action () method for associating actions with events, such as those on user interface components.

```
public boolean action(Event evt, Object arg) {
```

```
5    / /...
```

```
}
```

10 Typical behavior for an action () method is to get the data from the user interface event and send a message to the component of an application that is designated to handle it. In an ICE-T application, those messages are sent to the Presentation Engine using the sendMessage () method.

15 Here is an example of an action () method when in a GUI. It handles an event in which the user clicks a "Search" button after entering a text string'

```
public boolean action(Event evt, Object arg) {
```

```
20    //...
```

```
25    if (arg. equals ("Search"))
```

```
{
```

```
System.out.println("Search event is detected"); PeString firstName
= new PeString (entry_1.getText ()());
```

```
30    PeMessage msg = new PeMessage ("Search");
```

```
35    msg. addDataElement (firstName);
```

```
40    PeDebug.println("====> msg is: "+msg);
```

```
45    //send this event to the UI adaptor
```

```
40    pc.sendMessage (msg);
```

```
50    }
```

```
55    return true;
```

```
}
```

55 The action () method sends the message defined in:

PeMessage msg = new PeMessage("Search");

The PeMessage class encapsulates the message name and the data. The data in this message, added by the addDataElement () method, is a first name. The sendMessage () method sends the message and its data to the Presenta-

tion Engine through the PeUIAdaptor, a class that ICE-T provides for handling communication between the user interface and the Presentation Engine. Developers do not need to modify PeUIAdaptor, but do call methods in it as directed by the pe_template.

5 Handling Events from the Server

Events from the server program that generate messages for the client can be handled by:

- Updates to the user interface
- Updates to the Presentation Engine data model

10 Write a handler for each event from the server. Include it in the same file as the Presentation Engine, or a separate class file. If you use a separate file for the handler, remember to define it with public access.

15 Creating Handlers for Updates to the User Interface

To create a handler that updates the user interface, use the template to define a class that extends PeUIHandler. PeUIHandler is an abstract class for a handler of messages that are sent to the user interface.

20 The constructor method in PeUIHandler provides access to the uiContainer for updating the user interface and for sending messages using the uiAdaptor.

This example from the pe_template shows the definition for a class named SampleUIHandler. The constructor method passes adaptor and uiContainer as arguments to give your execute method access to them.

25 Use this definition as is, but supply the name of the class and add the body of the execute method:

```
25
class SampleUIHandler extends PeUIHandler {

30
    public SampleUIHandler(PeUIAdaptor adaptor, PeUI uiContainer) {
        super (adaptor, uiContainer);

    }

35
    public boolean execute (Object caller, PeMessage message) {

        //decode the record sent by the server

40
        //update the ui

        return true;
    }
}
```

50 To enable the Presentation Engine to know what messages to send to SampleUIHandler, register the name of the handler and the data it handles. This step is described in "Registering Handlers for Updates to the User Interface".

Creating Handlers for Updates to the Model

55 Messages from the server can be handled with code that updates the GUI (as in the example above), or code that stores the data in the Presentation Engine. If you used the createModel () method, you have to create handlers for the events that affect the model.

To create a handler that updates the model, use the template to define a class that:

- Extends PeModelHandler.

PeModelHandler is an abstract class for a handler of messages that are sent to the data model you defined with createModel().

5 • Uses the constructor of PeModelHandler as is, except to use the new class name in place of PeModelHandler.

public PeModelHandler(PeModel model) {}

This constructor passes the model class as an argument so that your Presentation Engine has access to the model class for updates.

10 This example from the pe_template shows the definition for a class named SampleModelHandler. Use this definition as is, but supply the name of the class and add the body of the execute method:

```
class SampleModelHandler extends PeModelHandler {
```

15

```
public SampleModelHandler (PeModel model) {  
    super (model);
```

20

```
}
```

25

public boolean execute (Object caller, PeMessage message) {

30

```
//Application code that decodes the record sent by the server
```

35

```
//Application code that updates the model
```

40

```
return true;
```

```
}
```

45

45 Registering Handlers

Register what component handles what messages in the Presentation Engine by filling in the createMessageHandler() method.

50 createMessageHandler() is defined in the pe_template to register handlers with either the user interface or the model. Use this method as defined, changing the arguments for adding handlers to supply the names of the messages and handlers you defined.

Registering Handlers for Updates to the User Interface

55 This code snippet from the Pe_template illustrates how to register the handler defined in the example in "Creating Handlers for Updates to the User Interface".

Note that you just use the code in the example as is. You don't need to specify anything other than the names of the messages and the objects that handle them. For each handler you register, add a call to the uiAdaptor.addHandler

() method. Change the arguments to the uiAdaptor.addHandler () method to specify the names of the messages and handlers you defined. In this example, the name of the message is "sample_ui_message" and the name of its handler is SampleUIHandler ():

```

5  protected void createMessageHandlers () throws
   DuplicateHandlerException {
10
15    uiAdaptor.addHandler ( "sample_ui_message",
                           new SampleUIHandler (uiAdaptor, uiContainer) );
20    //...

```

uiAdaptor and uiContainer are defined by the PresentationEngine class. Your handler needs to pass them as arguments so that its execute method has access to the user interface elements (GUI components).

Registering Handlers for Updates to the Model

If the message is to be handled in the Presentation Engine's data model, register the name of the handler in the model. This code is defined for you in the `pe_template`.

This code snippet illustrates how to register the handler defined in "Creating Handlers for Updates to the Model". Use the code in the example as is. Change the names of the messages and the methods that handle them:

```

30  protected void createMessageHandlers () throws DuplicateHandlerException
35  {
40
45    model.addHandler ( "sample_model_message",
                           new SampleModelHandler (model));
50    //...
55
60  }

```

45 Handling Shutdown Events

ICE-T provides default shutdown handlers. The shutdown handlers are defined in the `PresentationEngine` class, not in `pe_template`. Developers who do not want to accept the default shutdown handling can write their own shutdown handlers. A new handler takes precedence over the previous handler for the same message name. To add a new handler, developers:

- Write the handler
- Use the same message name ("ICET_SHUTDOWN")
- Use the methods provided for registering handlers in the Presentation Engine and the server program

When a shutdown happens, the Communication Library notifies the application by means of the shutdown han-

dlers. Shutdown handlers are identical to other message handlers, but have a preset message name ("ICET_SHUTDOWN"). To register a shutdown handler, the application code makes a call to addMessageHandler () using the preset message name.

5 Registering Shutdown Handlers in the Presentation Engine

An application can register two shutdown handlers in the Presentation Engine, one with the UI Adaptor and one with the model. Registering shutdown handlers in the Presentation Engine is optional.

To register a shutdown handler on the uiAdaptor, use the following call:

10 uiAdaptor.addHandler ("ICET_SHUTDOWN", new
shutdownHandlerUI (uiAdaptor, uiContainer));

To register a shutdown handler on the model, use the following call:

model.addHandler ("ICET_SHUTDOWN", new shutdownHandlerModel (model));

15 Registering Shutdown Handlers in the Server Program

Your application can register one shutdown handler with the server program.

To register a shutdown handler in the server program:

19 SrvComm_addMessageHandler ("ICET_SHUTDOWN",
20 <functionPointerToShutdownHandler>);

Provide the name of the function that executes when the shutdown occurs, (<functionPointerToShutdownHandler>) in the example above).

Preparing the Server Program for Communication

25 To prepare the server program to communicate with the client program, the developer "plugs in" the server program to ICE-T's Communication Layer. Connecting the server program to the server Communication Library, which is part of the ICE-T Communication Layer, is analogous to connecting the GUI to the Presentation Engine. In both cases, developers enable the exchange of messages, and these messages are handled locally by the components –the server program component and the Presentation Engine component.

30 The server program component in an ICE-T application must be written in any language that can call the C programming language. ICE-T provides both C and C++ language templates for handling messages and making calls to the server Communication Library. Use the appropriate template to start a new server program in C or C++, or add the template functions to an existing server program.

35 Note - if you have not done so already, copy one of the Templates subdirectories to the Application subdirectory. There are two subdirectories, one for C and one for C++. Each directory contains a server_template, one for C and the other for C++.

To enable communication between the server program and the client program:

40 • Create message handlers in the server program that are analogous to the message handlers you created in the Presentation Engine. This step is described in "Handling Messages in the Server Program".

• Make calls to the ICE-T server Communication Library.

45 The server program templates provide functions for message handling and communication. Developers just supply the application-specific message names and their handlers. For C server programs use server-template.C. For C++ programs use server_template.cc. The templates are in the ICE-T installation directory under Templates/C and Templates/C++ respectively.

50 Each server program template calls a default main routine. The default main () routines are provided for convenience. If you choose to write your own main () routine, look at default_main.c or default_main.cc for guidance and follow the steps in "Modifying the Default main Routine (Optional)".

55 Figure 28 describes the functions developers must fill in using the server program template. All three return integer values of 1 (TRUE) or 0 (FALSE). A return value of 1 (TRUE) indicates that the application startup can proceed. A return value of 0 (FALSE) indicates a problem that results in stopping the application. The remaining functions in the server program templates can be used as provided.

Figure 29 illustrates Server Properties in accordance with a preferred embodiment.

Handling Messages in the Server Program

In developing a Presentation Engine, a key task is writing the code that handles messages. The same task is performed for the server program. It consists of similar steps:

- 5 • Write the code that handles incoming events (typically requests for data)
- 10 • Register the event handlers
- 15 To handle messages in the server program:
 1. Write a function to handle each incoming message.
Each function you write takes one argument, a SrvMessage() function, and has a void return type.
The function should extract the data from the message, and then call application logic to process it or send a message back to the client.
 - 15 For example, here is a handler function in C that responds to a message with the data; employee name (empName) and employee number (empNumber):

20 **void handleEmployeeRecord (SrvMessage *message) {**

25 SrvData *empName;
 SrvData *empNumber;
 char *name;
 int num;

30 /*
 * disassemble incoming data
 */

35 empName= SrvMessage_getDataElement (message, 0);
 empNumber= SrvMessage_getDataElement (message, 1);

40

45

50

55

```

name=SrvString_getValue (empName);
num =SrvInteger_getValue(empNumber);

```

5

```

/*
 * Now process the data...
 */

```

10

```
lookupEmployee (name, number);
```

15

Here is the handler function in C++:

```

20 void handleEmployeeRecord (SrvMessage *message) {
/*
 * disassemble incoming data
 */
25 SrvData *empName= message->getDataElement (0);
SrvData *empNumber=message->getDataElement (1);

30 char *name=empName->getValue ( );
int num =empNumber->getValue();

35 /*
 * Now process the data...
 */
40

lookupEmployee (name, number);

```

45

2. Register the message handlers.

Fill in the `createMessageHandler ()` function in the `server-template` to register the handlers with the server Communication Library.

Note that you just use the code in the example as is. You don't need to specify anything other than the names of the messages and the functions that handle them. For each handler you register, add a `SrvComm_addMessageHandler ()` call. The arguments to the functions in this example are the application-specific names of the messages and handlers you defined in Step 1.

55

```

15
int createMessageHandlers () {
5   SrvComm_addMessageHandler ("Double", handleDouble);
  SrvComm_addMessageHandler ("firstNameEntered", handleSearchMessage);
  //...
  return 1;
10
}

```

15 **Modifying the Default main Routing (Optional)**

Once the server component of the application includes definitions for how it handles messages and registers those handlers, it can call the server Communication Library. Those calls are defined in the default main routines provided by ICE-T: default_main.c and default_main.cc.

20 default_main.c and default_main.cc are in the ICE-T installation directory under Server/Main.
 This section describes the order and usage of the calls to the server Communication Library.
 If you create your own main or modify the default, follow the instructions in this section for including the calls that enable the server program to work with the ICE-T Communication Layer.

25 To connect the server program to the Communication Library:

1. Include the ICE-T header files.

The default_main files include the header files already:

```

30 #include "SrvData.h"
  #include "SrvComm.h"
  #include "SrvMessage.h"

```

2. Call create_SrvComm ("tcp") to create the Communication Library's data structures.

```

35 char *protocol="tcp";
  create_SrvComm(protocol);

```

3. Call setSrvCommProperties (). (Optional) Set server properties. The setSrvCommProperties () function is typically implemented in a separate file that is linked to the server program as part of the build process.

40 4. Call createMessageHandlers () .

See "Handling Messages in the Server Program" for information about how to use createMessageHandlers () .

45 5. Call SrvComm_createSocket () to create the listener socket and output the port number.

Note - Do not write anything to stdout before you make the SrvComm_createSocket () call.

50 6. Initialize the application.

7. Call SrvComm_acceptClientConnection () to accept the connection from the client.

Note that the order of Step 7 and Step 8 may be interchanged.

55 8. Finally, call SrvComm_start () to start the Communication Library's loops.

This routine does not return until everything is finished.

Building ICE-T Applications

Building an ICE-T application consists of modifying a make file for the client and server programs, and then using

it to make both programs.

ICE-T provides a makefile (Example.mk) for both the Presentation Engine and the server programs. Example.mk is in the ICE-T installation directory under Templates/C or Templates/C++. These files are copied to the/Applications directory.

5 To use the makefile, modify it to specify the:

- Compiler location
- Libraries location
- ICE-T installation directory (optional)
- 10 • Client and server source file locations and names

1. Open Example.mk in an editor and follow the instructions in the file.
2. Supply the locations of the compiler, libraries. If you have moved the ICE-T installation directory, supply its new location.

15 There is no need to change the location of the ICE-T installation directory. If you choose to move it, supply the name of the location in the TCET-INSTALL - DIR macro.

The macros for which you provide values are shown below as indicated in the code:

```

20 ##########
##### Environment Configuration #####
#####

25 #
# Change the following lines to indicate the location of your compiler.
#
30 COMPILER_BIN=
COMPILER_LIB=
#
35 # Change the following line to indicate the location of the ICE-T installation
# directory (either absolute or relative to the current directory).
# The default value (../..) allows you to build an example in-situ in the
40 # installation hierarchy.
#
45 ICET - INSTALL - DIR=../..

```

3. Change the macros for the Presentation Engine source files. This example specifies the Java files for the Presentation Engine template (pe_template.java) and a user interface file named myGui.java. The macros for which you provide values are shown here in bold type. Change the names of the files to those used in your application:

50

55

```

#####
###      PE      ###
5  ###  (Configurable Macros)###
#####

10 #
# Change the following macro to add your java files
#
15

PE_SOURCES5.java= \
20      myGui.java \
      pe_template.java \
      #end
25 #
# Change the following macro to indicate which of the java classes # is the top
# level class (i.e. the subclass of presentationEngine) .
# Note: specify this class without any extension, e.g.:
30 # PE_MAIN_CLASS=myPresentationEngine
#
35 PE_MAIN_CLASS=pe_template

```

40 Example.mk specifies files for the server program template (server_template). The macros for which you provide values are shown here in bold type. There is a macro for C source files and one for C++ source files. Change the names of the files to those used by the server program in your application:

45

50

55

```

#####
###      Server      ###
5  ###  (Configurable Macros)  ###
#####

10  #
# change the following macro to add .cc (C Plus Plus) files
#
15  SERVER_SOURCES.cc=\
    #end
#
20  # change the following macro to add .c (C) files
#
25  SERVER_SOURCES.c=\
    default_main.c \
    server_template.c \
    #end
#
35  # change the following macro to indicate the name of the
# server executable
SERVER = ServerTemplate

40
    Build the client program with the following command:
    % make -f Example.mk pe
    Build the server program:
45  % make -f Example.mk server
    Both pe and server are defined for you in the Makefile.

```

Testing ICE-T Applications

50 You can test the Presentation Engine alone or with the server program. You have to build the programs before you test them.

- Run the Presentation Engine class by itself using the java command.

55 For example:
 % java myPresentationEngine
 To test the Presentation Engine with the server program:

1. After you have made the server program, run it to get the port id.

This example shows the result of running a server program named myServer:

```
5 % myServer
// ICE LOG messages deleted from this example.
37526 129.146.79.147
ICE: :ApplicationServer: port=37526 ServerIP=129.146.79.147
```

2. Supply the server machine IP and the port id as arguments to the java command.

For example, using the machine and port id from Step 1:

```
10 % java myPresentationEngine 129.146.79.147 37526
```

3. To test the Presentation Engine in debug mode, use the -debug option to the java command.

The - debug option should be the first argument after the class name:

```
15 % java myPresentationEngine - debug 129.146.79.147 37526
```

Configuring and Deploying ICE-T Applications

In addition to tools for developing a Presentation Engine, ICE-T provides these components to support application deployment, execution, and administration:

Installation scripts

The following installation scripts are in the ICE-T installation directory under/bin:

- ice-httdp-setup

Sets up the directories you need on the server.

- ice-install-access

Installs the Access Layer on the server.

- ice-app-install

Installs each compiled Presentation Engine and server program on the server.

- Access Layer

Acts as gatekeeper for HTTP communication between the client and server nodes. The Access Layer is a cgi-bin executable that identifies and authenticate users. You can modify the e files used by the Access Layer and then use the supplied makefile (Access.mk) to build a customized Access program for use with ICE-T server applications.

- Application startup applet template (Java)

A template for making Java applets that launch ICE-T applications. The template is in the ICE-T installation directory under StartApplet.

Web server (user must install)

Supports HTTP connections; enables Web access and the use of a browser front end for executing an ICE-T application.

Note - Sun internal users can install the Web server from /home/internet/CERN/httdp/. The directory contains a README file with installation instructions.

Deploying and maintaining ICE-T applications involves these steps:

1. Using a stamp applet and HTML pages to launch ICE-T applications

2. Setting up the Web server

3. Customizing (optional) and installing the Access Layer

4. Running installation scripts for Presentation Engines and server programs

5. Configuring application management files

Using Startup Applets and HTML Files

Compiled Presentation Engines can run as applets in a Java-enabled browser. To enable users to launch ICE-T applications from a browser use the named ICE-T templates to:

- 5 • Create a startup applet for each application. Use the startAppletDevIR.java template.
"Using the Startup Applet" describes this step.
- 10 • Create a top-level HTML file with links to each application. This file serves as a "splash page" identifying the applications available and including links to an HTML file for each application. Use splashTemplate.html.
"Creating a Top-level HTML File" describes this step.
- 15 • Create an HTML file for each application. Use appTemplate.html.
"Creating Individual Application HTML Files" describes how.

Using the Startup Applet

A startup applet provides a way to launch an ICE-T application from a Web page. The startup applet:

- 20 • Starts the HTTP access to the server
- Authenticates the user
- Starts the server program
- Downloads the Presentation Engine as an applet to the Web page or opens the applet in a separate user interface window (developer's choice)

25 You can use the startAppletDevIR.java template to launch the applications you build, or create your own applet. The template is completely generalized so that it can be used for any of your applications. You supply the application name in a separate parameter to the applet tag. (See "Creating Individual Application HTML Files".)

A complete code listing for startAppletDevIR.java is in Appendix B.

- 30 By default, the startup applet opens the Presentation Engine in the same window. To have the Presentation Engine appear in a new browser window, open startAppletDevIR.java and follow the instruction in the file:

```
AppletContext cont = app.getAppletContext();
35    if (cont != null) {
        System.out.println ("Showing document: ["+newURL+"] ");
        cont.showDocument (newURL);
        /*
        * To show the applet in another window, replace the
        * the line above with the commented line below.
        */
        //cont.showDocument (newURL, "new_window");
    }
50
```

Creating a Top-Level HTML File

- 55 You need a top-level HTML file, or "splash page," to present the list of available applications with links to the application-level HTML pages for them. When a user chooses a link, the link takes them to an application-level HTML page. ICE-T provides a default HTML file for the "splash page." The file, called splashTemplate.html, is in the ICE-T installation directory under StartApplet. You can use the default file or make one of your own.

To create a top-level Web page for listing the links to your application Web pages:

1. Copy `splashTemplate.html` to another file. For example:

`% cp splashTemplate.html myAppSplashPage.html`

5 2. Open the file in an editor.

10 3. Provide a title for the page and any text you want to include about the application(s) listed there.

15 4. Supply the name and the URL of the application-level HTML page for each listed application.

For example, if you used `appTemplate.html` to create an HTML file for an application named `MyApplication`:

15 `<html>`

16 •

17 •

18 •

20 ``

25 `MyApplication1`

``

26 •

27 •

28 •

30 `</html>`

35

40 5. Copy the file to the appropriate location on your Web server.

The result is a Web page with a link to the named application:

`MyApplication1`

45 When a user chooses this link, the browser loads an application-level HTML page with the startup applet for the application.

46 **Creating Individual Application HTML Files**

50 Think of the application-level HTML file as the Presentation Engine startup page. This page contains the startup applet that results in the Presentation Engine being downloaded.

When the user launches the stamp applet, the startup applet takes the name of the application from the applet parameter, the user name, and the password and sends these using HTTP to the Access Layer on the server.

To create an application-level Web page:

55 1. Copy `aiDIDTemplate.html` to another file. For example:

`% cp appTemplate.html myAppPage.html`

2. Open the file in an editor.

3. Include instructions to the user on how to launch the application. (Optional)

startAppletDevIR.java defines a Send button and a class (sendBtn) that handles the user's input. appTemplate.html includes default instructions for using Send. If you want to change the user's interaction with the stamp applet, you would need to change the sendBtn class and the instructions in appTemplate.html.

5 4. Specify the name of the startup applet.

If you have copied startAiDiDletDevIR.java, given it another name, and compiled it, supply that name to the applet tag:

```
(applet code="startAppletDevIR.class" width=400 height=400)
```

10 5. Provide the application name as a parameter to the applet tag. For example:

```
(param name=AppName value = "MyApplication!")
```

15 6. Provide the name of the access program to use for this application. The default value is "Access." You may have customized the default access file and given it another name. If so, provide the name as a value to the Access parameter.

```
(param name=Access value="Access")
```

Be sure that the file you create from appTemplate.html contains the end applet tag (/applet).

20 Here are the tags for a minimal HTMIL file using startAppletDevIR, an application need "MyApplication!", and the default access program name:

```
(html)
(blockquote)
Please provide Username and Password and press the "Send" button to launch the application.
/blockquote>


---


(applet code="startAppletDevIR.class" width=400 height=400)
(param name=AppName value="MyApplication!" (param name=Access value="Access"))
/applet)
/html)
```

25 When the user launches the startup applet and sends the user data and the application name to the server, ICE-T's Access Layer:

- 35 • Authenticates the user name and password
- Downloads the Presentation Engine as an applet in a Web page

As part of the process of determining access and downloading Presentation Engines, the Access Layer relies on an application configuration file to provide information about the location and names of program components. ICE-T's

40 Access Layer installation script generates the application configuration file automatically. That configuration is the basis for generating an HTML wrapper file in which to download the Presentation Engine. You can accept the defaults and let your application use the generated HTML wrapper, or you can customize the application configuration file so that it generates a customized HTML file to hold the Presentation Engine. See "Configuring Applications" for more information.

45 Setting up the Web Server

Note - Complete the following Web server setup before installing the ICE-T application, that is, before running ice-app-install.

50 This step establishes a Web server file structure for the ICE-T files that manage access to and delivery of client applications. To establish this file structure, run the script provided in the ICE-T installation directory under /bin/ice-htpd-setup.

Before you run this server setup script, you should establish links to the cgi-bin directory and the httpd-docs directory.

55 Create a symbolic link to the cgi-bin and httpd-docs directories using the UNIX ln command.

- Become superuser
- Make the links:

% in -s (absolute path to cgi-bin)/cgi-bin
 % in -s (absolute path to httpd-docs)/WWW-docs

If you cannot create these links, provide the absolute path names to cgi-bin and httpd-docs as arguments to the ice-httdp-setup script. If you have not created the links, you will need to change the Access Layer's default access properties file to specify the cgi-bin and httpd-docs locations. (See "Customizing the Access Layer"). If you have created the links, run ice-httdp-setup without arguments.

From the ICE-T installation directory, run ice-httdp-setup.ice-httdp-setup takes these arguments:

- -cgi-bin – takes the location of the cgi-bin directory.
 The default is/cgi-bin.
- -httpd-docs – takes the location of the httpd/docs directory.
 The default is/www-docs.

15 Note - Run ice-httdp-setup once for each machine that is an ICE-T Web server.

For example:

% cd (ICE-T installation directory)
 % ./bin/ice-httdp-setup
 ice-httdp-setup establishes a Web server file structure for the files that manage access to and delivery of requested client applications.

ice-httdp-setup performs these tasks:

- Creates an ICE directory under the -cgi-bin directory
- Creates a link to cgi-bin/ICE from the httpd-docs directory
- Creates an application startup directory under the ICE directory
- Creates an ICEAppRepository directory

ice-httdp-setup takes additional optional arguments. Run the setup file with the -help qualifier to get a list of the

30 arguments and their defaults:

% cd (ICE-T installation directory)
 % ./bin/ice-httdp-setup -help

Customizing the Access Layer

35 This step is optional. The Access Layer uses authentication and access properties files to manage access to ICE-T applications. Developers can use the files provided by ICE-T or modify them to specify their own access requirements or to create a number of access subsystems for use with different applications.

The Access Layer uses:

- default_authentication.cc
 default_authentication.cc contains a single default authentication function that checks the application name, user name, and password. By default, all users are authenticated. Modify the file to specify how to authenticate users.
- default_access_properties.cc
 default_access_properties contains several functions for controlling access log file properties, as well as functions for specifying the Access directory, and password scrambling keys.

50 With the exception of the cgi_bin_location property, (See table below) no modification is necessary for these files, but you may choose to change the defaults for authentication and access properties. The files are in the Access subdirectory in the ICE-T installation directory.

The table appearing below describes properties that can be modified in default_access_properties.cc:

Table Default Access Properties

Property	Default	Description
cgi_bin_location	/cgi-bin	If you have not created a link to cgi-bin (as described in "Setting up the Web Server", provide a cgi-bin location here.
log_destination	logfile	The return value determines where log entries are written.
logging_level	all	Accepted values: console, logfile. If you choose to write the log to a file, set the file location and name using the logfile property.
logfile	(cgi_bin_location)/ICE/Access logs/ Access - pid.log	Determines to what level the Access Layer logs access attempts. Accepted value: none, all, starts, errors, debug.
scrambler_key	Uses the scrambler key provided by ICE-T.	Return value is used as a key to unscramble the user name and password sent when the user launches an application. If you change the scrambler key, change the scrambler key in the startup applet also.
sessionEnvironment		Sets up the calling environment for the server program. You can make calls to putenv() to set environment variables whose values can be accessed with getenv() from the server program.

To implement the authenticate() function in default_authentication.cc:

1. copy the file from the ICE-T installation's Access directory.
2. Follow the instructions in the file to implement the authenticate() function.
3. In Access.mk, replace this file name with yours.

To change the defaults in default_access_properties.cc:

1. Copy the file from the ICE-T installation's Access directory and give it a name that suits the application.
2. Modify the file to supply application-specific values.
3. Edit Access.mk to replace the default file names with the files you edited. Access.mk is in the Access directory. The user-modifiable section is marked.

To substitute the new file name for default_authentication.cc, change the second line:

```
AUTHENTICATION_SOURCES.cc = \
default_authentication.cc \
```

To substitute the new file name for default_access_properties.cc, change the second line:

```
ACCESS_PROPERTIES_SOURCES.cc = \
default_access_properties.cc \
```

4. Run Access.mk.

```
% make -f Access.mk
```

5. After you configure the Access Layer, run ice- install -access.

By default, Access .ink creates a an executable named CustomAccess. If you have changed the default name of the access program, use the - source option to ice-install-access and specify the name you have given the access program. For example:

5 % ice-install-access -source myAccess

Installing the Access Program

The ICE-T Access Program installation script:

10 • Installs the Access program on the server in the cgi -bin directory that you specify

14 • Installs the startup applets in the/www-docs/ICE/start directory.

To install the Access program, run ice- install -access from the ICE-T installation directory:

15 % cd (ICE-T installation directory)

16 %/bin/ice- install-access

ice-install-access takes four optional arguments:

20 • -cgi-bin (cgi-bin location)

The default is/cgi -bin.

25 • -source (Access executable name)

The name of the Access program that you are copying from. The default is Access.

30 • -dest (Access executable name)

The name of the Access program that you are copying to. The default is Access. Change the destination if you want to use different access programs for different applications.

35 • -help

36 Displays the arguments and their defaults.

Installing the ICE-T Application

ICE-T provides a script for installing applications.

35 ice- app- install performs these tasks:

40 • Installs the client and server executable files for the application

45 • Creates an application directory

50 • Installs the server libraries

55 • Installs the Java packages used by the Java Presentation Engine class

60 • Creates an application configuration file (appConfigFile), or installs a specified one

If you already have an appconfigFile to which you have made changes, you can avoid having the application installation overwrite it. Use the - appConfigFile (file)option.

65 • Copies the aIDIDConfigFile to the application directory

70 Installing the ICE-TApplication with ice-app-install

Note - Run ice-app-install for each application you want to install.

From the ICE-T installation directory, run ice-app-install with the required arguments:

75 • -appName – the name of the application

80 • -peClass – the name of the Presentation Engine class

- -peSource – the location of the Presentation Engine you want to install
- -serverName – the name of the server program
- 5 • -serverSource -- the location of the server program you want to install
If peSource and serverSource are the same directory, you only need to specify one of them.

ice - app- install takes additional optional arguments: ice- app- install is in the /bin directory of the ICE-T application installation directory. You can run the file with the -help option to get a list of the arguments and their defaults:

10 % cd {ICE-T installation directory}
%/bin/ice- app- install -help

Installing the ICE-T Application with the Provided Makefile

15 An alternative to running ice- app- install, is to use the Example.mk Makefile to install the completed application. ExamIDle.mk has defined a target for installing the application on your Web server. If you have set up the Web server, creating the symbolic links as described in "Setting up the Web Server" then you can use ExamIDle.mk with the following argument:
% make -f Example.mk app-install

20 If you did not create a symbolic link to the cgi-bin directory as described in "Setting up the Web Server", specify the cgi-bin location in the Example.mk file (use the CGI_BIN_LOCATION macro and then run make on the aIDID - install target;
% make -f Example.mk app-install

25 **Configuring Applications**

The ICE-T application installation script (ice-app- install) generates a default application configuration file (appConfigFile). When a user starts an application by launching a startup applet, the Access Layer uses the Application Manager to look up values for server and client program locations and names in appConfigFile. Using the configuration file, 30 the Application Manager generates an HTML wrapper for presenting Presentation Engines as applets in a Web browser execution environment. (See "Using Startup Applets and HTML Files" for more information about how to use startup applets for ICE-T applications.)

To complete the last step in deploying an application to a Web Browser you use one of two ways to supply application-specific values in the HTML wrapper:

35 • Run ice - app- install with the required arguments as described in "Installing the ICE-T Application" and let it create the default aIDIDConfigFile.

• Or, create your own application configuration file by modifying the automatically generated appConfigFile.

40 By default, ice - app-install creates the application file in the /cgi-bin/ICE/ICEAppRepository directory and names it (appName).appconf.

To customize the appConfigFile generated by ice-app- install:

45 1. Open the generated configuration file ((appName).appconf) in an editor.

2. Supply the names for each application in the appropriate tags in each appConfigFile.
You are required to supply application information in these tags:

50 • (peClass)-- the name of the Presentation Engine class

• (serverName)-- the name of the server program

• (PresentationEngine)-- specifies where the Presentation Engine appears on the Web page.
55 The Application Manager replaces the (Presentation Engine) tag with a Java (applet) tag that specifies the named Presentation Engine (peClass).

3. Supply messages to return to the browser if user authentication or application stamp fails.

The template contains tags for authentication failure and application startup failure messages.

The appConfigFile contains optional tags for you to specify a Web page title, headings, and text, just as you would for any Web page.

5 The Application Manager also uses a properties file to set what applications are returned to the client. By default, the Application Manager returns the application name passed to it by the Access Layer. You can specify another application to return by modifying default_appmgr_properties.cc. default_appmgr_properties.cc is described in "Customizing the Access Layer".

10 **Presentation Engine Template**

ICE-T provides a null application that you can use as a template for Presentation Engines. You can find the file for the template in the ICE-T application installation directory under /Templates/pe_template.java.

15 **Code Example A-1 pe_template.java Listing**

```
import sunsoft.ice.pe.*;
20 import java.net.*;
import java.io.*;
import java.applet.*;
25 import java.util.*;
import java.awt.*;

30 /*
 * This is a sample template of the ICE-T Presentation Engine - it needs to
 * be filled with the actual ui, names of messages and handlers to create
35

40

45

50
```

55

```

* a working PE.
*/
5 // Extend the PresentationEngine class
public class pe_template extends PresentationEngine {
// Constructor
10 public pe_template() {
    // This is required. It sets up the internal PE components
    super();
15 }

```

Code Example A-1 pe_template.java Listing (Continued)

```

20 /**
* This is called when the current applet terminates.
25 * It needs to be customized for handling the termination
* of the current applet.
*/
30 public void terminate (String reason) {
    super.PEtermiate(reason);
}
35 /**
* This can be used to do any local initializations that may be required in
40 * the PE. It is called before all the connections are setup with the server
* so no messages should be sent from this function.
*/
45 protected void initializeApplication() {
}
50 /**
* createUI creates the UI components of the applet.
* It uses the "ui" object of "gui" class which is generated
* by the SpecJava gui builder.

```

```
5      */
10     /**
11      * Specify the UI of the application in this function.
12      */
13     protected void createUI() {
14
15         // create local awt components to map the ones generated
16         // by specJava. For example:
17
18
20
21     /**
22      * createModel Creates the data items in the Model. Some data items
23      * can be observable. These items which have an associated observer
24      * function that gets called when the observable is updated.
25      */
26
27     // This function is optional. It is only needed if the
28     // application wants to do some client-side application logic
29     // Uncomment the lines below if you plan to do client side processing
30     // in the Model.
31
32
33
34
35
```

Code Example A-1 pc_template.java Listing (Continued)

```
45 protected void createModel() {  
46     //create the observable object e.g. PeHashtable, PeVector, PeString . . .)  
47  
48     //Attach observer to the observable.  
49  
50     //put the observables into Model
```

```
5
10
15
20
25
30
35
40
45
50
55
  /**
   * provide the mapping of Message names to handlers for the inbound
   * Messages to the PE from the UI or Comm adaptors.
   * The Handlers for the UI and Model messages should
   * be provided in this function.
   */
// This function is required. It keeps the mapping of UI events
// registered with the uiAdaptor and the model events registered
// with the model
protected void createMessageHandlers() throws DuplicateHandlerException {
    // UI maps
    uiAdaptor.addHandler ("sample_ui_message",
    new SampleUIHandler (uiAdaptor, uiContainer));
    // ...
    // Model Maps
    model.addHandler ("sample_model_message",
    new SampleModelHandler(model));
    // ...
}

/*
 * main is used when running as a standalone java application,
 * (i.e. not in a browser environment), and is intended to do

```

* all the things that the browser would do for the applet.

5 Code Example A-1 pe_template.java Listing (Continued)

```
*  
10 */  
  
15       public static void main(String args[]) {  
  
20        pe_template pe = new pe_template();  
25        pe.isApplet = false;  
30        pe.getCommandLineArgs (args);  
35        pe.init(); pe.start();  
40        }  
45        }  
50        }
```

55 ||||||||||||||||| UI Message Handlers
60 |||||||||||||||||

```
65        class SampleUIHandler extends PeUIHandler {  
70        public SampleUIHandler( PeUIAdaptor adaptor, PeUI uiContainer) {  
75        super (adaptor, uiContainer);  
80        }  
85        public boolean execute (Object caller, PeMessage message) {  
90        // decode the record send by the server  
95        // update the ui  
100        return true;  
105        }
```


5

Startup Applet Template

10 startAppletDevIR.java is a Java applet that launches ICE-T applications. The file is generalized to run any ICE-T application. You don't need to change it unless you want to change the user's interaction with the applet. The applet currently asks for username and password and provides a Send button to launch (start up) the application. The file is in the ICE-T installation directory under/StartApplet. For instructions on how to use this file, see "Using Startup Applets and HTML Files".

15

Code Example B-1 startAppletDevIR.java Listing

```

20 /**
 * This is a sample applet that can be used for gathering specific
 * user information before displaying the ICE-T application. It can
 * be customized as needed. It is loaded into the browser using the
25 * index.html file provided.
 */
30
35 import java.net.*;
import java.io.*;
import java.applet.*;
import java.util.*;
import java.awt.*;
import sunsoft.ice.ui.*;
40
45 public class startAppletDevIR extends Applet {

```

45

Code ExampleB-1 startAppletDevIR.java Listing (Continued)

50

55

```

public startAppletDevIR () {
}

5

public void init () {

10    setLayout (new ColumnLayout ());
    Panel fnamep = new Panel ();
    fnamep.setLayout (new FlowLayout (FlowLayout.LEFT));
    fnamep.add (new Label ("Username:"));
    TextField fnameText= new TextField("",20);
    fnamep.add(fnameText);
    20    add(fnamep);

    Panel passwdp = new Panel ();
    passwdp.setLayout (new FlowLayout (FlowLayout.LEFT));
    passwdp.add (new Label("Password: "));
    TextField passwdText= new TextField("",20);
    25    passwdText.setEchoCharacter(',');
    passwdp.add (passwdText);
    add(passwdp);

30

    Panel btmp = new Panel ();
    btmp.setLayout (new FlowLayout ());
    AButton cbtn = new Abutton ("Send");

    cbtn.setAction (new sendBtn ());
    cbtn.addClientData ("applet",this);
    35    cbtn.addClientData ("username",fnameText);
    cbtn.addClientData ("password",passwdText);
    btmp.add (cbtn);
    add(btmp);

50

55

```

```

show ();
}

5

public String getAppName() {
    appname = getParameter ("AppName");
10
    return appname;
}

15

public String getAccessName () {
    String acc = getParameter ("Access");

```

20 **Code ExampleB-1 startAppletDevIR.java Listing (Continued)**

```

if (acc == null || acc.equals("")) acc="Access";
25
return acc;
}

30

public String getAppHost () {
    System.err.println ("Applet: "+isApplet);
    if (isApplet) {
35
        URL docBase=getDocumentBase ();
        System.err.println("Document base: "+docBase);

40
        apphost=docBase.getHost ();
        if (apphost==null) apphost="";
        int port=docBase.getPort ();
        if (port != -1) apphost = apphost + ":"+port;
        System. err.println("Server is: "+apphost+"");
45
    } else {
50
        apphost="";

```

55

```

    }

    return apphost;
}

public static void main (String args[]) {
    10   Frame f = new Frame ("ICE-T startup applet");

    startAppletDevIR tap = new startAppletDevIR ();
    15   tap.isApplet=false;
    tap.init (); f.add ("Center", tap);
    f.resize (500,400);
    20   f.show ();
    tap.start ();
}

25

public boolean isApplet=true;
private String appname = new String("");
private String apphost = new String("");
}

30

35  /*
 * sendBtn handles the "Send" button activation. The execute
 * member of this class is called when the "Send" button is
 * 40  pressed. It collects all the relevant user information and

```

Code ExampleB-1 startAppletDevIR.java Listing (Continued)

```

45
 * sends it to the Server via HTTP. This is done by executing
 * the Access function in the server's cgi-bin.
50
 * [Note: the username and password is scrambled so it will not
 *      be displayed in the URL field of the browser.]
 */

55

```

```

class sendBtn extends Activity {
    public boolean execute (Object caller, Event event) {
5       AButton btn= (AButton) caller;
       startAppletDevIR app = (startAppletDevIR) (btn.getClientData ("applet"));
       TextField name = (TextField) (btn.getClientData
10      ("username"));
       TextField passwd = (TextField) (btn.getClientData
("password"));
15      String servername = app.getAppHost ();

       URL newURL = null;
20      String username=name.getText ();
       String userpasswd=passwd.getText ();
       String appname = app.getAppName ();
25      String accessName=app.getAccessName ();
       AccessScrambler scrambler=new AccessScrambler ();
       System.err.println ("Scrambling ("+username+", "+
30      //userpasswd+", "+
      appname+");

       String scrambled=scrambler.scrambleUser (username, userpasswd, appname);
35      System.err.println("  scrambled ==> ["+scrambled+"]");

       String url ="http://" + servername +
40      "/cgi-bin/" +accessName + "?" + appname +
      "+ " + scrambled;

45      System.err.println (url);

       try {
50      newURL = new URL(url);
      }
      catch (Exception e) {

```

```

System.err.println ("Exception Getting newURL from (" +url+ "): " +e);
}

```

```

5
}

```

```

AppletContext cont = app.getAppletContext();
if (cont != null) {

```

```

10

```

Code ExampleB-1 startAppletDevIR.java Listing (Continued)

```

15

```

```

/*
 * The call to ShowDocument makes the httpd request to the Access
 * cgi-bin executable.

```

```

20

```

```

* Call showDocument with one argument (the URL) to show the
* PE in a single window; call with two arguments (the URL, "new_window")
* to show the PE in a new browser window.

```

```

25

```

```

* We're currently defaulting the call to bring up a new browser
* window because of the way browsers handle backing out of html
* pages with java applets — if you go too far back, then the applet
* is destroyed. We put the PE in a new window to minimize the chance
* of going too far back simply because you want your browser back,
* but don't intend to kill your PE.

```

```

30

```

```

* /
35
System.err.println ("Showing document in new window: [" +newURL+ "] in
window: (" +appname+ ")");

```

```

40

```

```

cont.showDocument (newURL, appname);

```

```

//cont.showDocument (newURL);

```

```

//System.err.println ("Showing document: [" +newURL+ "]");

```

```

45

```

```

}

```

```

50
return true;
}

```

```

55

```

5

Server Program Templates

This appendix contains code listings for the following templates:

- 10 • server_template.c
- default_main.c
- 15 • server_template.cc
- default_main.cc

20 Chapter 2, "Building Program Components" describes the location and use of these templates. See "Handling Messages in the Server Program" and "Modifying the Default main Routine (Optional)" for more information.

C++ Files

25 ICE-T provides a server program template and a default main routine for application developers using C++ to develop server programs.

C++ Server Program Template

Code Example C-1 server_template.cc Listing

```
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
125
130
135
140
145
150
155
160
165
170
175
180
185
190
195
200
205
210
215
220
225
230
235
240
245
250
255
260
265
270
275
280
285
290
295
300
305
310
315
320
325
330
335
340
345
350
355
360
365
370
375
380
385
390
395
400
405
410
415
420
425
430
435
440
445
450
455
460
465
470
475
480
485
490
495
500
505
510
515
520
525
530
535
540
545
550
555
560
565
570
575
580
585
590
595
600
605
610
615
620
625
630
635
640
645
650
655
660
665
670
675
680
685
690
695
700
705
710
715
720
725
730
735
740
745
750
755
760
765
770
775
780
785
790
795
800
805
810
815
820
825
830
835
840
845
850
855
860
865
870
875
880
885
890
895
900
905
910
915
920
925
930
935
940
945
950
955
960
965
970
975
980
985
990
995
1000
1005
1010
1015
1020
1025
1030
1035
1040
1045
1050
1055
1060
1065
1070
1075
1080
1085
1090
1095
1100
1105
1110
1115
1120
1125
1130
1135
1140
1145
1150
1155
1160
1165
1170
1175
1180
1185
1190
1195
1200
1205
1210
1215
1220
1225
1230
1235
1240
1245
1250
1255
1260
1265
1270
1275
1280
1285
1290
1295
1300
1305
1310
1315
1320
1325
1330
1335
1340
1345
1350
1355
1360
1365
1370
1375
1380
1385
1390
1395
1400
1405
1410
1415
1420
1425
1430
1435
1440
1445
1450
1455
1460
1465
1470
1475
1480
1485
1490
1495
1500
1505
1510
1515
1520
1525
1530
1535
1540
1545
1550
1555
1560
1565
1570
1575
1580
1585
1590
1595
1600
1605
1610
1615
1620
1625
1630
1635
1640
1645
1650
1655
1660
1665
1670
1675
1680
1685
1690
1695
1700
1705
1710
1715
1720
1725
1730
1735
1740
1745
1750
1755
1760
1765
1770
1775
1780
1785
1790
1795
1800
1805
1810
1815
1820
1825
1830
1835
1840
1845
1850
1855
1860
1865
1870
1875
1880
1885
1890
1895
1900
1905
1910
1915
1920
1925
1930
1935
1940
1945
1950
1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
2025
2030
2035
2040
2045
2050
2055
2060
2065
2070
2075
2080
2085
2090
2095
2100
2105
2110
2115
2120
2125
2130
2135
2140
2145
2150
2155
2160
2165
2170
2175
2180
2185
2190
2195
2200
2205
2210
2215
2220
2225
2230
2235
2240
2245
2250
2255
2260
2265
2270
2275
2280
2285
2290
2295
2300
2305
2310
2315
2320
2325
2330
2335
2340
2345
2350
2355
2360
2365
2370
2375
2380
2385
2390
2395
2400
2405
2410
2415
2420
2425
2430
2435
2440
2445
2450
2455
2460
2465
2470
2475
2480
2485
2490
2495
2500
2505
2510
2515
2520
2525
2530
2535
2540
2545
2550
2555
2560
2565
2570
2575
2580
2585
2590
2595
2600
2605
2610
2615
2620
2625
2630
2635
2640
2645
2650
2655
2660
2665
2670
2675
2680
2685
2690
2695
2700
2705
2710
2715
2720
2725
2730
2735
2740
2745
2750
2755
2760
2765
2770
2775
2780
2785
2790
2795
2800
2805
2810
2815
2820
2825
2830
2835
2840
2845
2850
2855
2860
2865
2870
2875
2880
2885
2890
2895
2900
2905
2910
2915
2920
2925
2930
2935
2940
2945
2950
2955
2960
2965
2970
2975
2980
2985
2990
2995
3000
3005
3010
3015
3020
3025
3030
3035
3040
3045
3050
3055
3060
3065
3070
3075
3080
3085
3090
3095
3100
3105
3110
3115
3120
3125
3130
3135
3140
3145
3150
3155
3160
3165
3170
3175
3180
3185
3190
3195
3200
3205
3210
3215
3220
3225
3230
3235
3240
3245
3250
3255
3260
3265
3270
3275
3280
3285
3290
3295
3300
3305
3310
3315
3320
3325
3330
3335
3340
3345
3350
3355
3360
3365
3370
3375
3380
3385
3390
3395
3400
3405
3410
3415
3420
3425
3430
3435
3440
3445
3450
3455
3460
3465
3470
3475
3480
3485
3490
3495
3500
3505
3510
3515
3520
3525
3530
3535
3540
3545
3550
3555
3560
3565
3570
3575
3580
3585
3590
3595
3600
3605
3610
3615
3620
3625
3630
3635
3640
3645
3650
3655
3660
3665
3670
3675
3680
3685
3690
3695
3700
3705
3710
3715
3720
3725
3730
3735
3740
3745
3750
3755
3760
3765
3770
3775
3780
3785
3790
3795
3800
3805
3810
3815
3820
3825
3830
3835
3840
3845
3850
3855
3860
3865
3870
3875
3880
3885
3890
3895
3900
3905
3910
3915
3920
3925
3930
3935
3940
3945
3950
3955
3960
3965
3970
3975
3980
3985
3990
3995
4000
4005
4010
4015
4020
4025
4030
4035
4040
4045
4050
4055
4060
4065
4070
4075
4080
4085
4090
4095
4100
4105
4110
4115
4120
4125
4130
4135
4140
4145
4150
4155
4160
4165
4170
4175
4180
4185
4190
4195
4200
4205
4210
4215
4220
4225
4230
4235
4240
4245
4250
4255
4260
4265
4270
4275
4280
4285
4290
4295
4300
4305
4310
4315
4320
4325
4330
4335
4340
4345
4350
4355
4360
4365
4370
4375
4380
4385
4390
4395
4400
4405
4410
4415
4420
4425
4430
4435
4440
4445
4450
4455
4460
4465
4470
4475
4480
4485
4490
4495
4500
4505
4510
4515
4520
4525
4530
4535
4540
4545
4550
4555
4560
4565
4570
4575
4580
4585
4590
4595
4600
4605
4610
4615
4620
4625
4630
4635
4640
4645
4650
4655
4660
4665
4670
4675
4680
4685
4690
4695
4700
4705
4710
4715
4720
4725
4730
4735
4740
4745
4750
4755
4760
4765
4770
4775
4780
4785
4790
4795
4800
4805
4810
4815
4820
4825
4830
4835
4840
4845
4850
4855
4860
4865
4870
4875
4880
4885
4890
4895
4900
4905
4910
4915
4920
4925
4930
4935
4940
4945
4950
4955
4960
4965
4970
4975
4980
4985
4990
4995
5000
5005
5010
5015
5020
5025
5030
5035
5040
5045
5050
5055
5060
5065
5070
5075
5080
5085
5090
5095
5100
5105
5110
5115
5120
5125
5130
5135
5140
5145
5150
5155
5160
5165
5170
5175
5180
5185
5190
5195
5200
5205
5210
5215
5220
5225
5230
5235
5240
5245
5250
5255
5260
5265
5270
5275
5280
5285
5290
5295
5300
5305
5310
5315
5320
5325
5330
5335
5340
5345
5350
5355
5360
5365
5370
5375
5380
5385
5390
5395
5400
5405
5410
5415
5420
5425
5430
5435
5440
5445
5450
5455
5460
5465
5470
5475
5480
5485
5490
5495
5500
5505
5510
5515
5520
5525
5530
5535
5540
5545
5550
5555
5560
5565
5570
5575
5580
5585
5590
5595
5600
5605
5610
5615
5620
5625
5630
5635
5640
5645
5650
5655
5660
5665
5670
5675
5680
5685
5690
5695
5700
5705
5710
5715
5720
5725
5730
5735
5740
5745
5750
5755
5760
5765
5770
5775
5780
5785
5790
5795
5800
5805
5810
5815
5820
5825
5830
5835
5840
5845
5850
5855
5860
5865
5870
5875
5880
5885
5890
5895
5900
5905
5910
5915
5920
5925
5930
5935
5940
5945
5950
5955
5960
5965
5970
5975
5980
5985
5990
5995
6000
6005
6010
6015
6020
6025
6030
6035
6040
6045
6050
6055
6060
6065
6070
6075
6080
6085
6090
6095
6100
6105
6110
6115
6120
6125
6130
6135
6140
6145
6150
6155
6160
6165
6170
6175
6180
6185
6190
6195
6200
6205
6210
6215
6220
6225
6230
6235
6240
6245
6250
6255
6260
6265
6270
6275
6280
6285
6290
6295
6300
6305
6310
6315
6320
6325
6330
6335
6340
6345
6350
6355
6360
6365
6370
6375
6380
6385
6390
6395
6400
6405
6410
6415
6420
6425
6430
6435
6440
6445
6450
6455
6460
6465
6470
6475
6480
6485
6490
6495
6500
6505
6510
6515
6520
6525
6530
6535
6540
6545
6550
6555
6560
6565
6570
6575
6580
6585
6590
6595
6600
6605
6610
6615
6620
6625
6630
6635
6640
6645
6650
6655
6660
6665
6670
6675
6680
6685
6690
6695
6700
6705
6710
6715
6720
6725
6730
6735
6740
6745
6750
6755
6760
6765
6770
6775
6780
6785
6790
6795
6800
6805
6810
6815
6820
6825
6830
6835
6840
6845
6850
6855
6860
6865
6870
6875
6880
6885
6890
6895
6900
6905
6910
6915
6920
6925
6930
6935
6940
6945
6950
6955
6960
6965
6970
6975
6980
6985
6990
6995
7000
7005
7010
7015
7020
7025
7030
7035
7040
7045
7050
7055
7060
7065
7070
7075
7080
7085
7090
7095
7100
7105
7110
7115
7120
7125
7130
7135
7140
7145
7150
7155
7160
7165
7170
7175
7180
7185
7190
7195
7200
7205
7210
7215
7220
7225
7230
7235
7240
7245
7250
7255
7260
7265
7270
7275
7280
7285
7290
7295
7300
7305
7310
7315
7320
7325
7330
7335
7340
7345
7350
7355
7360
7365
7370
7375
7380
7385
7390
7395
7400
7405
7410
7415
7420
7425
7430
7435
7440
7445
7450
7455
7460
7465
7470
7475
7480
7485
7490
7495
7500
7505
7510
7515
7520
7525
7530
7535
7540
7545
7550
7555
7560
7565
7570
7575
7580
7585
7590
7595
7600
7605
7610
7615
7620
7625
7630
7635
7640
7645
7650
7655
7660
7665
7670
7675
7680
7685
7690
7695
7700
7705
7710
7715
7720
7725
7730
7735
7740
7745
7750
7755
7760
7765
7770
7775
7780
7785
7790
7795
7800
7805
7810
7815
7820
7825
7830
7835
7840
7845
7850
7855
7860
7865
7870
7875
7880
7885
7890
7895
7900
7905
7910
7915
7920
7925
7930
7935
7940
7945
7950
7955
7960
7965
7970
7975
7980
7985
7990
7995
8000
8005
8010
8015
8020
8025
8030
8035
8040
8045
8050
8055
8060
8065
8070
8075
8080
8085
8090
8095
8100
8105
8110
8115
8120
8125
8130
8135
8140
8145
8150
8155
8160
8165
8170
8175
8180
8185
8190
8195
8200
8205
8210
8215
8220
8225
8230
8235
8240
8245
8250
8255
8260
8265
8270
8275
8280
8285
8290
8295
8300
8305
8310
8315
8320
8325
8330
8335
8340
8345
8350
8355
8360
8365
8370
8375
8380
8385
8390
8395
8400
8405
8410
8415
8420
8425
8430
8435
8440
8445
8450
8455
8460
8465
8470
8475
8480
8485
8490
8495
8500
8505
8510
8515
8520
8525
8530
8535
8540
8545
8550
8555
8560
8565
8570
8575
8580
8585
8590
8595
8600
8605
8610
8615
8620
8625
8630
8635
8640
8645
8650
8655
8660
8665
8670
8675
8680
8685
8690
8695
8700
8705
8710
8715
8720
8725
8730
8735
8740
8745
8750
8755
8760
8765
8770
8775
8780
8785
8790
8795
8800
8805
8810
8815
8820
8825
8830
8835
8840
8845
8850
8855
8860
8865
8870
8875
8880
8885
8890
8895
8900
8905
8910
8915
8920
8925
8930
8935
8940
8945
8950
8955
8960
8965
8970
8975
8980
8985
8990
8995
9000
9005
9010
9015
9020
9025
9030
9035
9040
9045
9050
9055
9060
9065
9070
9075
9080
9085
9090
9095
9100
9105
9110
9115
9120
9125
9130
9135
9140
9145
9150
9155
9160
9165
9170
9175
9180
9185
9190
9195
9200
9205
9210
9215
9220
9225
9230
9235
9240
9245
9250
9255
9260
9265
9270
9275
9280
9285
9290
9295
9300
9305
9310
9315
9320
9325
9330
9335
9340
9345
9350
9355
9360
9365
9370
9375
9380
9385
9390
9395
9400
9405
9410
9415
9420
9425
9430
9435
9440
9445
9450
9455
9460
9465
9470
9475
9480
9485
9490
9495
9500
9505
9510
9515
9520
9525
9530
9535
9540
9545
9550
9555
9560
9565
9570
9575
9580
9585
9590
9595
9600
9605
9610
9615
9620
9625
9630
9635
9640
9645
9650
9655
9660
9665
9670
9675
9680
9685
9690
9695
9700
9705
9710
9715
9720
9725
9730
9735
9740
9745
9750
9755
9760
9765
9770
9775
9780
9785
9790
9795
9800
9805
9810
9815
9820
9825
9830
9835
9840
9845
9850
9855
9860
9865
9870
9875
9880
9885
9890
9895
9900
9905
9910
9915
9920
9925
9930
9935
9940
9945
9950
9955
9960
9965
9970
9975
9980
9985
9990
9995
9999
```

```

#include <stdio.h>
#include <string.h>
5 #include <sys/types.h>

// these are required to get the server side ICE-T components
10

#include "SrvData.hh"
#include "SrvComm.h"
15 #include "SrvMessage.hh"

#define false 0
20 #define true 1

/*****
25 * The Message Handler functions *
*****/
30 /* This is the Message handler function that gets called when a specified
   * message is received from the client. It has only one argument
   * (msg) that contains the message name and message data.
35 */
40 void handleMessage (SrvMessage *msg) {
  /*
   * disassemble incoming data
   */

```

45 Code Example C-1 server_template.cc Listing (Continued)

```

45 /*
50 SrvData *data;
data=message->getDataElement (0);

```

55

```

*/  

5  /*  

 * The application "Logic" */  

*/  

10  

/*  

 * Assemble outgoing data  

*/  

15  

// send a reply if necessary. <replyData> can be of the types  

20 // provided in SrvData.hh  

/*  

SrvMessage *reply=new SrvMessage ("replyMessageName");  

25 reply->adclDataElement (<replyMessageData>);  

SrvComm_sendMessage (reply);  

*/  

30 }  

35 //This is an example of a shutdown handler on the server  

void shutdownHandler (SrvMessage *shutdownMsg) {  

40 char *shutdownType= ((SrvString *) shutdownMsg->getDataElement (0)) -  

>getValue ();  

char *reasonString= ((SrvString *) shutdownMsg->getDataElement (1))->getValue  

45 0;  

if (!strcmp (shutdownType,"ICET_CLIENT_SHUTDOWN")) {  

50 fprintf (stderr,"shutdownHandler: Detected CLIENT SHUTDOWN event.  

Reason=%s\n", reasonString);  

} else if (!strcmp (shutdownType,"ICET_CLIENT_DISCONNECT")) {  

55

```

```

sprintf (stderr,"shutdownHandler: Detected CLIENT DISCONNECT event.
Reason=%s\n", reasonString);
5   } else if (!strcmp(shutdownType,"ICET_SERVER_SHUTDOWN")) {
sprintf (stderr,"shutdownHandler: Detected SERVER SHUTDOWN event.
Reason=%s\n", reasonString);
10  } else if (!strcmp (shutdownType,"ICET_INTERNAL_SHUTDOWN")) {

```

Code Example C-1 server_template.cc Listing (Continued)

```

15
sprintf (stderr,"shutdownHandler: Detected INTERNAL SHUTDOWN event.
Reason=%s\n", reasonString);
20  } else {
sprintf (stderr,"Shutdown Handler: unknown msg type (%s). Reason=%s\n",
shutdownType, reasonString);
25  }
}

30 ****
*
* Functions required by default-main, that set the
* SrvComm library properties (setSrvCommProperties) and
35  * initialize the applicationService (InitializeApplicationService)
*
****

40 int setSrvCommProperties () {
/* The server timeout is how long the server waits between receiving
* messages from the PE before it initiates the timeout shutdown procedure.
45  */
SrvComm_setServerTimeout (300); /* in seconds */

50 /* The accept timeout is how long the server will wait to accept a connection
* from the client before it shuts down.

```

55

```

*/  

SrVComm_setAcceptTimeout (400); /* in seconds */  

5  

return 1;  

}  

10  

int createMessageHandlers () {  

SrVComm_addMessageHandler ("SampleIncomingMessage", handleMessage);  

15 SrVComm_addMessageHandler ("ICET_SHUTDOWN", shutdownHandler);  

return 1; // (return 0; if there is a problem here)  

20 }
  

int initializeApplication () {  

25  

return 1; // (return 0; if there is a problem initializing the application)  

}  

30

```

Default main for C++

35 **Code Example C-2 default_main.cc Listing**

```

/*****  

40 * default_main.cc  

*  

* This file implements the default main () used  

45 * by an ICE-T server executable.  

*  

* The structure of this main is that it calls  

50 * all the functions necessary to run  

* the ICE-T SrvCom library, and provides several  

* places for the application writer to include  

55

```

- * application-specific code. These functions
- * are implemented by the application-writer,
- * and are usually implemented in a separate file, that gets
- * linked in as part of the build process.
- *
- 10 * These functions all return int values, which are used to determine if
- * the startup of the application should continue or should be aborted.
- * In this way the application code has a hook for indicating when something
- 15 * goes wrong (can't connect to database, or something), and passing this
- * information on to the application so startup doesn't continue.
- * Return 1 (TRUE) to indicate that everything is OK, or return 0 (FALSE)
- 20 * to indicate a fatal problem that should result in aborting the application
- startup.
- *
- 25 * int setSrvComProperties ()
- * Fill in this function to set properties on the SrvComm library.
- * This function is called after the SrvCom library is created, but before it accepts
- 30 * a connection from the client.
- *
- 35 * int createMessageHandlers ()
- * Fill in this function to register the message handlers.
- * This function is called immediately after setSrvComProperties () .
- *
- 40 * int initializeApplication ()
- * Fill in this function to start up the application-specific code.
- * It is called after the connection is accepted from the client,
- * and before the message reader and handler loops are started.
- 45 *
- *
- * Code Example C-2 default_main.cc Listing (Continued)
- *
- 50 * NOTES:
- * This main () routine is provided for your convenience,

- * but you are not required to use it. You are
- * free to write your own main () routine, provided
- * you follow the requirements for ordering the
- * calls to the SrvComm library:
 - * 1. First call create-SrvComm("tcp"); to create
* the Comm library's data structures.
 - * 2. Set the properties on the Comm library and
* register message handlers and other handlers.
 - * 3. Call SrvComm_createSocket () to create the
* listener socket and output the port number
* (Note: do NOT write anything to stdout before
* making this call).
 - * 4. Initialize the application.
 - * 5. Call SrvComm_acceptClientConnection () to
* accept the connectio fromthe client.
* (Note: steps 4 and 5 may be interchanged)
 - * 6. Finally call SrvComm_start () to start the
* Comm library's loops. This routine does not
* return until everything is finished.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#include "SrvComm.h"

extern int setSrvCommProperties ();
extern int createMessageHandlers ();
extern int initializeApplication ();
```

```
*****
*.. The default Main
*
* ICE-T provides a default main() routine
* for ICE-T server applications.
*
*****/
```

15 Code Example C-2 default_main.cc Listing (Continued)

```
int
20 main(
    int argc,
    char** argv,
    25 char** envp
)
{
30
    char *protocol="tcp";

35 create_SrvComm(protocol);

40 /*
    * Call user-defined function to set the properties on the Comm Library.
    * Then call user-defined function to create the Message handlers
    * and add them to the Comm library.
    *
45    * The setSrvCommProperties function is where Comm Library properties
    * are set. All properties have reasonable defaults, so none of them are
    * required to be set.
    *
50
55
```

* The `createMessageHandlers` function is where the message handlers are
 * added, and other handlers (shutdown handler, default message handler,
 5 * etc) are registered.
 *
 * Requirements on these two functions:
 10 * 1. These functions may NOT write anything to stdout, since the Access
 * layer expects the first thing on stdout to be the port number that
 * the server will be listening for a client connection on.
 15 *
 */
 if (!setSrvCommProperties ()) {
 20 exit (0);
 }
 25 if (!createMessageHandlers ()) {
 exit(0);
 }
 30 /*
 * Create the socket that the client will connect to.
 35 * This routine also is responsible for generating (and printing to

Code Example C-2 default_main.cc Listing (Continued)

40 * stdout) the port number so the Access layer can generate the
 * html code that passes the port number to the client applet
 * (presentation engine).
 45
 *
 50 * (NOTE: Do not write anything to stdout before this routine is called.
 * Writing to stderr is OK, however.)
 */

```

if (SrvComm_createSocket () != SRVCOMM_STATUS_OK) {
    exit(0);
5
}

10   sprintf (stderr,"ICE::ApplicationServer: port=%d
ServerIP=%s\n",SrvComm_getPort (), SrvComm_getIPAddress ());
    sprintf (stderr,"\\n\\n\\n\\n");
    fflush(stderr);

15

16  /*
17   * call user-defined function to initialize the application
18   *
19   * Requirements on this function:
20   *   1. This function initializes any data structures in the application
21   *   that need to be initialized before incoming messages
22   *   from the client get handled.
23   *   2. This function must return. This means that if the application has
24   *   its own notifier loop that, say, listens to messages from a database,
25   *   then that notifier loop must be called in a new thread.
26   */
27
28   if (!initializeApplication ()) {
29     exit(0);
30   }

35

36  /*
37   * Accept a connection from the client.
38   * (blocks until a connect attempt is made, or until the
39   * acceptTimeout expires).
40   */
41
42   if (SrvComm_acceptClientConnection () != SRVCOMM_STATUS_OK) {
43     exit(0);
44   }

45
46
47
48
49
50
51
52
53
54
55

```

1*
 5 * Finally, we start up the server loops.
 * This function does not return until everything is finished.

10 Code Example C-2 default_main. cc Listing (Continued)

15 */
 SrvComm_start();
 }
 20 /**
 25 * This function is called when the server times out.
 * It returns TRUE or FALSE (int 1 or 0), TRUE indicating that the
 * timeout-shutdown should proceed; FALSE indicating that no, don't
 * shutdown on this timeout (the timeout timer gets reset).
 *
 30 */
 */ #ifdef __cplusplus
 extern "C" {
 35 #endif
 int handleServerTimeout () {
 return 1;
 }
 40 #ifdef __cplusplus
 }
 45 #endif

50

55

C Server Program Template

Code Example C-3 server-template.c Listing

```

5      /**
6      *server-template.c
7      *
8      *ICE-T server template.
9      *
10     */
11
12     #include    <stdio.h>
13     #include    <string.h>
14     #include    <sys/types.h>
15
16
17     #include    "SrvData.h"
18     #include    "SrvComm.h"
19
20     #include    "SrvMessage.h"
21
22     #define false 0
23     #define true 1
24
25     /*This is the Message handler function that gets called when a specified *
26     message is received from the client. It has only one argument * (msg) that
27     contains the message name and message data.
28
29     */
30
31     void handleMessage(SrvMessage *message) {
32
33     /*
34     *disassemble incoming data
35     */
36
37     /*
38     SrvData *data; data=SrvMessage->getDataElement (message, 0);
39
40     */
41
42     CodeExample C-3 server-template.c Listing (Continued)
43
44     * The application "Logic" ,/
45     Assemble outgoing data ,/

```

```

– see SrvData.h for how to create data types. --SrvData ,replyData; replyData=
create-SrvVector();
5 SrvVector-addElement(replyData, create-SrvString("Something")); ./
/
Create the SrvMessage struct ,/
10 /reply=create-SrvMessage("replyMessage");
SrvMessage-addDataElement(replyData); ./
/*
15 *Send message ./
/*SrvComm-sendMessage(reply); ./
/*This is an example of a shutdown handler on the server */
void shutdownHandler(SrvMessage ,shutdownMsg) {
20 SrvData ,shutdownTypeSStr= SrvMessage-getDataElement(shutdownMsg, 0);
SrvData *reasonSStr= SrvMessage-getDataElement(shutdownMsg, 1); char
25 *reasonString=SrvString-getValue(reasonSStr);
Code Example C-3 server-template. c Listing (Continued)
char *shutdownType=SrvString-getValue(shutdownTypeSStr);
if (strcmp(shutdownType,"ICET-CLIENT-SHUTDOWN")) {
30 fprintf(stderr,"shutdownHandler: Detected CLIENT SHUTDOWN event.
Reason=%s\n", reasonString);
} else if (strcmp(shutdownType,"ICET-CLIENT-DISCONNECT")) {
35 fprintf(stderr,"shutdownHandler: Detected CLIENT DISCONNECT event.
Reason=%s\n", reasonString);
} else if (!strcmp(shutdownType,"ICET-SERVER-SHUTDOWN")) {
40 fprintf(stderr,"shutdownHandler: Detected SERVER SHUTDOWN event.
Reason=%s\n", reasonString);
} else if (strcmp(shutdownType,"ICET-INTERNAL-SHUTDOWN")) {
45 fprintf(stderr,"shutdownHandler: Detected INTERNAL SHUTDOWN event.
Reason=%s\n", reasonString);
} else {
50 fprintf(stderr,"Shutdown Handler: unknown msg type (%s). Reason=%s\n",
shutdownType, reasonString);
}

```

55

```

5
}
}

10 ****
*
* Functions required by default-main, that set the
* SrvComm library properties (setSrvCommProperties) and
* initialize the applicationService (initializeApplicationService)
*
15 ****
/
```

```

15
int setSrvCommProperties() {
    /*The server timeout is how long the server waits between receiving
    *messages from the PE before it initiates the timeout shutdown procedure.
    */SrvComm-setServerTimeout(300); /*, in seconds */
    /*The accept timeout is how long the server will wait to accept a connection */
    *from the client before it shuts down.
    */SrvComm-setAcceptTimeout(400); /* in seconds */
    return 1;
}

30
int createMessageHandlers()
{ SrvComm-addMessageHandler("SampleIncomingMessage", handleMessage);
  SrvComm-addMessageHandler("ICET-SHUTDOWN", shutdownHandler);
  return 1;
}

35
int initializeApplication() {
  return 1;
}

40
}

45

```

Default mainfbr C

Code Example C-4 default-main.c Listing

5 *****
 default-main.c

10 This file implements the default main() used by an ICE-T server executable.

15 The structure of this main is that it calls
 all the required functions necessary to run

20 the ICE-T SrvComm library, and provides several places for the application
 writer to include application-specific code. These functions are required to be
 implemented by the application-writer, and are usually implemented in a
 separate file, that gets linked in as part of the build process.

25 These functions all return int values, which are used to determine if the startup
 of the application should continue or should be aborted.

30 In this way the application code has a hook for indicating when something goes
 wrong (can't connect to database, or something), and passing this information
 on to the application so startup doesn't continue.

35 Return 1 (TRUE) to indicate that everything is OK, or return 0 (FALSE)
 to indicate a fatal problem that should result in aborting the application startup.

40 int setSrvCommProperties()

45 Fill in this function to set properties on the SrvComm library. This function is
 called after the SrvComm library is created, but before it accepts a connection
 from the client.

50 int createMessageHandlers()

Fill in this function to register the message handlers.

5 This function is called immediately after setSrvCommProperties().

10 int initializeApplication()

15 Fill in this function to start up the application-specific code. It is called after the connection is accepted from the client, and before the message reader and handler loops are started.

15 NOTES:

20 This main() routine is provided for your convenience,

25 CodeExample C-4 default-main.c Listing (Continued)

25 but you are not required to use it. You are free to write your own main() routine, provided you follow the requirements for ordering the calls to the SrvComm library:

30 1. First call create-SrvComm("tcp"); to create the Comm library's data structures.

35 2. Set the properties on the Comm library and register message handlers and other handlers.

40 3. Call SrvComm-createSocket() to create the listener socket and output the port number (Note: do NOT write anything to stdout before * making this call).

45 4. Initialize the application.

45 5. Call SrvComm-acceptClientConnection() to accept the connection from the client. (Note: steps 4 and 5 may be interchanged)

50 6. Finally call SrvComm-start() to start the Comm library's loops. This routine does not return until everything is finished.

```

*****  

5      #include      <stdio.h>  

6      #include      <stdlib.h>  

7      #include      <string.h>  

10     #include      <sys/types.h>  

11     #include "SrvComm.h"  

15     extern int setSrvCommProperties();  

16     extern int createMessageHandlers();  

17     extern int initializeApplication();  

20  

*****  

21     *      The default Main  

22     *  

23     *      ICE-T provides a default main() routine  

24     *      for ICE-T server applications.  

25  

30  

*****  

31     int  

32  

33     main(  

34  

35         int argc,  

36         char** argv,  

40         char** envp  

41  

42     )  

43  

44     {  

45  

46         char ,protocol="tcp";  

50         create-SrvComm(protocol);  

55         /*  


```

5 *Call user-defined function to set the properties on the Comm Library. * Then
 call user-defined function to create the Message handlers * and add them to the
 Comm library.

10 The setSrvCommProperties function is where Comm Library properties are set.
 All properties have reasonable defaults, so none of them are required to be set.

15 The createMessageHandlers function is where the message handlers are added,
 and other handlers (shutdown handler, default message handler, etc) are
 registered.

20 *Requirements on these two functions:

25 *1. These functions may NOT write anything to stdout, since the Access ,
 layer expects the first thing on stdout to be the port number that *
 the server will be listening for a client connection on.

 *

30 */if (:setSrvCommProperties()) {

35 exit(0);
 }

40 if (!createMessageHandlers()) {
 exit(0);

45 }
 /*

50 * Create the socket that the client will connect to.

55

*This routine also is responsible for generating (and printing to
 *stdout) the port number so the Access layer can generate the
 5 *html code that passes the port number to the client applet

CodeExample C-4 default-main.c LLstmg (Continued)

10 *(presentation engine).

*

15 *(NOTE: Do not write anything to stdout before this routine is called.
 *Writing to stderr is OK, however.)

20 */

if (SrvComm-createSocket() != SRVCOMM-STATUS-OK) {
 25 exit(0);

 fprintf(stderr, "ICE::ApplicationServer: port=%d
 ServerIP=%s\n", SrvComm-getPort(), SrvComm-getIPAddress());
 30 fprintf(stderr, "\n\n\n\n");
 fflush(stderr);

35 /*

* call user-defined function to initialize the application

40 *

*Requirements on this function:

45 * 1. This function initializes any data structures in the application
 * that need to be initialized before messages incoming messages
 * from the client get handled.

50

55

- * 2. This function must return. This means that if the application has its own notifier loop that, say, listens to messages from a database, then that notifier loop must be called in a new thread.

```

5
  */
10
  if (!initializeApplication()) {
    exit (0);
15
  }
20
  /*.
   *Accept a connection from the client.
   *(blocks until a connect attempt is made, or until the
   *acceptTimeout expires).
25
   */
30
  /*.
   *Finally, we start up the server loops.
   *This function does not return until everything is finished.
35
  SrvComm-start();
}
40
  /**
45
   *This function is called when the server times out.
   *It returns TRUE or FALSE (int 1 or 0), TRUE indicating that the
   *timeout-shutdown should proceed, FALSE indicating that no, don't
   *shutdown on this timeout (the timeout timer gets reset).
50
  */
55
  #ifdef __cplusplus
  extern "C"

```

```

5      #endif

6      int handleServerTimeout() {
7          return 1;
8
9      }

10     #ifdef __cplusplus
11     #endif

```

15

ICE-T Exceptions Catalog D

ICE-T client program exceptions are caught by the modules in the Presentation Engine. The `IctExceptionHandler` generates exception messages. It issues warnings and errors by printing messages to the Java console in a Netscape Navigator environment, or to the terminal where Java was started (in the case of the Java development kit).

Server (Communication Library) exceptions are caught by the modules in the Communication Layer. The `ServerExceptionHandler` generates messages. It issues warnings and errors by printing messages to the application logfile if the application is launched from a browser or to stdout if the application is run standAlone.

25 Here is a sample message:

ICET WARNING: handled in PresentationEngine.init

Attempt to register duplicate handlers.

Figure 30 is a table of client and server side exceptions in accordance with a preferred embodiment.

While the invention is described in terms of preferred embodiments in a specific system environment, those skilled in the art will recognize that the invention can be practiced, with modification, in other and different hardware and software environments within the spirit and scope of the appended claims.

Claims

35 1. A server for a distributed system, comprising:

- (a) a client computer;
- (b) a server computer;
- (c) a network connecting the client computer to the server computer;
- 40 (d) an execution framework code segment configured to couple the server computer and the client computer via the network, comprising:

- (1) a plurality of client computer code segments resident on the server, each for transmission over the network to a client computer to initiate coupling;
- (2) a plurality of server computer code segments resident on the server which execute on the server in response to initiation of coupling via the network with a particular client utilizing the transmitted client computer code segment for communicating via a particular communication protocol;

- 45 (e) the client computer code segment including a mediator state machine which receives a plurality of messages, determines which message should be handled by which part of the execution framework, and forwards the message for further processing to the execution framework; and
- (e) the execution framework dispatches messages and initiates events in response to characteristics of the message transferred by the mediator state machine.

55 2. The server for a distributed system as recited in claim 1, including a user interface part of the client code segment which receives messages containing user interface information and translates the user interface information into display information.

3. The server for a distributed system as recited in claim 1, including a communication adaptor for communicating message types bound for external consumption to the network.

5. The server for a distributed system as recited in claim 1, including a web connection which facilitates communication from the client computer to the server computer.

10. The server for a distributed system as recited in claim 1, including a communication protocol corresponding to the client code segment and the server code segment which is supported by a communication library on the client computer and the server computer.

15. The server for a distributed system as recited in claim 1, including a code segment which authenticates an initial request from a client computer before downloading a client code segment and initiating secure communication.

20. The server for a distributed system as recited in claim 4, wherein the web is the Internet.

25. A method for distributing computing between a server computer system and a client computer system coupled by a network, comprising the steps of:

- (a) coupling the server computer and the client computer through the network utilizing an execution framework code segment, comprising:
 - (1) a plurality of client computer code segments resident on the server, each for transmission over the network to a client computer to initiate coupling;
 - (2) a plurality of server computer code segments resident on the server which execute on the server in response to initiation of coupling via the network with a particular client utilizing the transmitted client computer code segment for communicating via a particular communication protocol;
- (b) receiving a plurality of messages at the client computer code segment including a mediator state machine;
- (c) determining which message should be handled by which part of the execution framework at the mediator state machine;
- (d) forwarding the message for further processing to the execution framework; and
- (e) dispatching messages and initiating events in response to characteristics of the message transferred by the mediator state machine.

35. 9. The method as recited in claim 8, including the step of invoking a user interface part of the client code segment to receive messages containing user interface information and translating the user interface information into display information.

40. 10. The method as recited in claim 8, including the step of communicating message types bound for external consumption to the network utilizing a communication adaptor.

45. 11. The method as recited in claim 8, including the step of utilizing a web connection which facilitates communication from the client computer to the server computer.

50. 12. The method as recited in claim 8, including the step of utilizing a communication protocol corresponding to the client code segment and the server code segment which is supported by a communication library on the client computer and the server computer.

55. 13. The method as recited in claim 8, including the step of authenticating an initial request from a client computer before downloading a client code segment and initiating secure communication.

14. The method as recited in claim 11, wherein the web is the Internet.

15. A computer program embodied on a computer-readable medium for enabling a distributed computer system, comprising:

- (a) a code segment for responding to a request from a client computer system to a server computer system; and

(b) an execution framework code segment configured to couple the server computer and the client computer via the network, comprising:

5 (1) a plurality of client computer code segments resident on the server, each for transmission over the network to a client computer to initiate coupling;

10 (2) a plurality of server computer code segments resident on the server which execute on the server in response to initiation of coupling via the network with a particular client utilizing the transmitted client computer code segment for communicating via a particular communication protocol;

15 (c) the client computer code segment including a mediator state machine which receives a plurality of messages, determines which message should be handled by which part of the execution framework, and forwards the message for further processing to the execution framework; and

20 (d) the execution framework including code that dispatches messages and initiates events in response to characteristics of the message transferred by the mediator state machine.

16. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 15, including a user interface part of the client code segment which receives messages containing user interface information and translates the user interface information into display information.

25 17. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 15, including a communication adaptor for communicating message types bound for external consumption to the network.

25 18. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 15, including a web connection which facilitates communication from the client computer to the server computer.

30 19. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 15, including a communication protocol corresponding to the client code segment and the server code segment which is supported by a communication library on the client computer and the server computer.

35 20. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 15, including a code segment which authenticates an initial request from a client computer before downloading a client code segment and initiating secure communication.

35 21. The computer program embodied on a computer-readable medium for enabling a distributed computer system as recited in claim 18, wherein the web is an Internet.

40

45

50

55

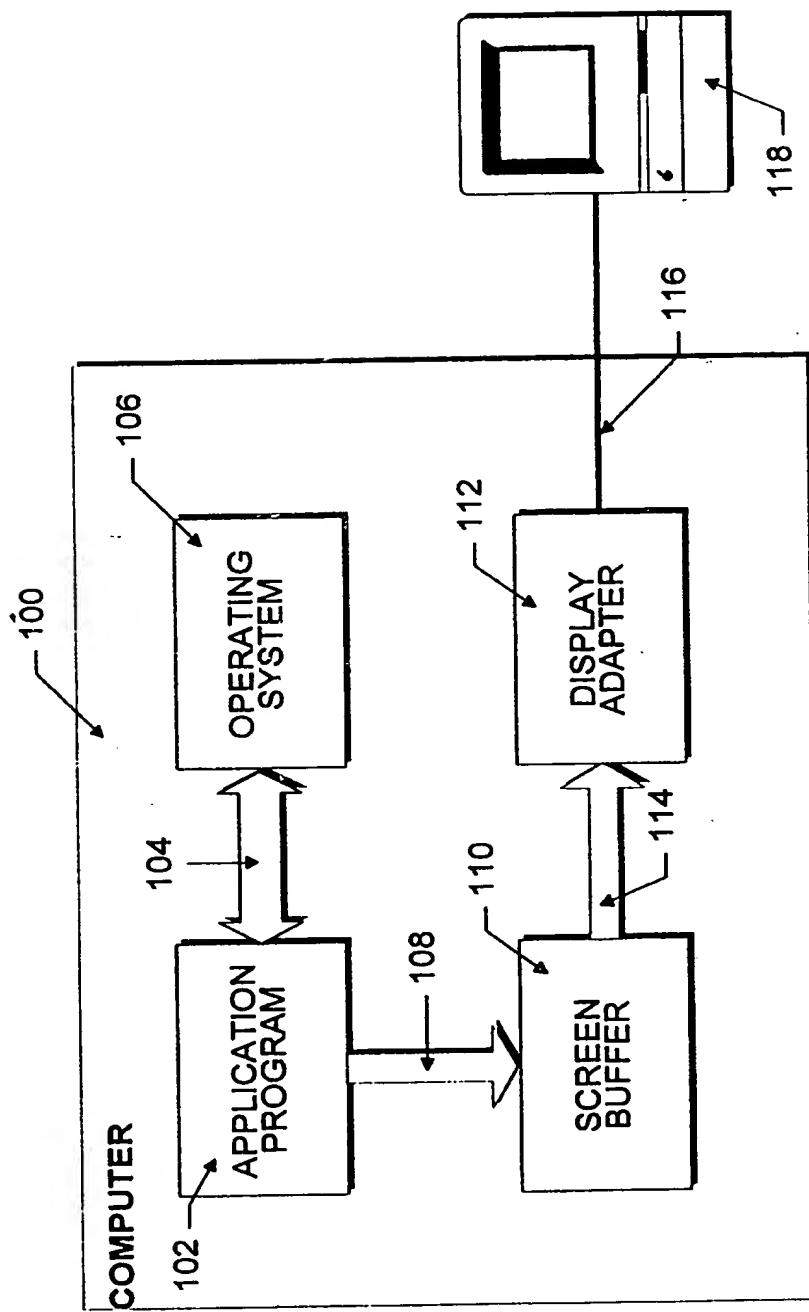


FIG.-1
(PRIOR ART)

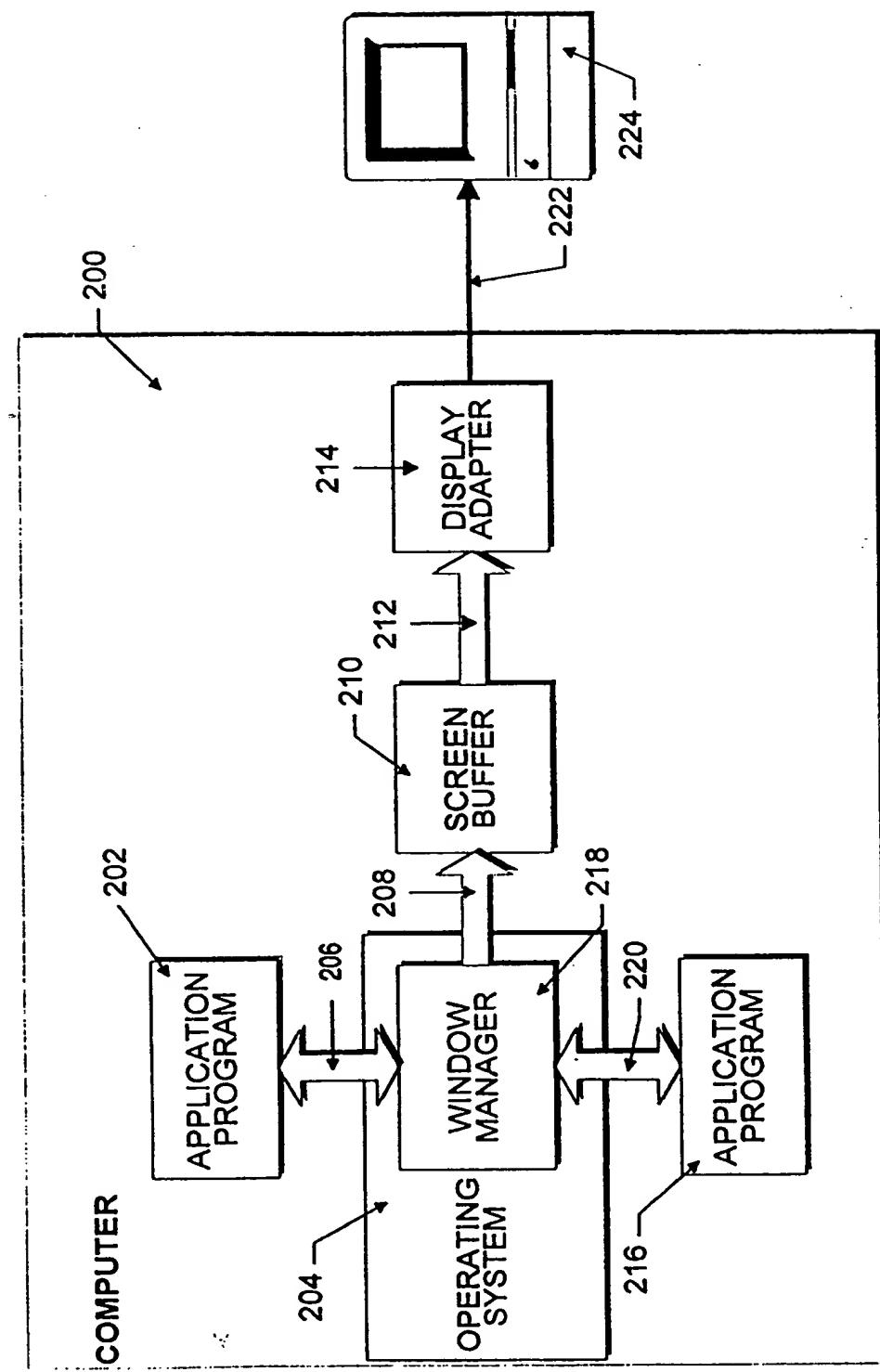


FIG. -2
(PRIOR ART)

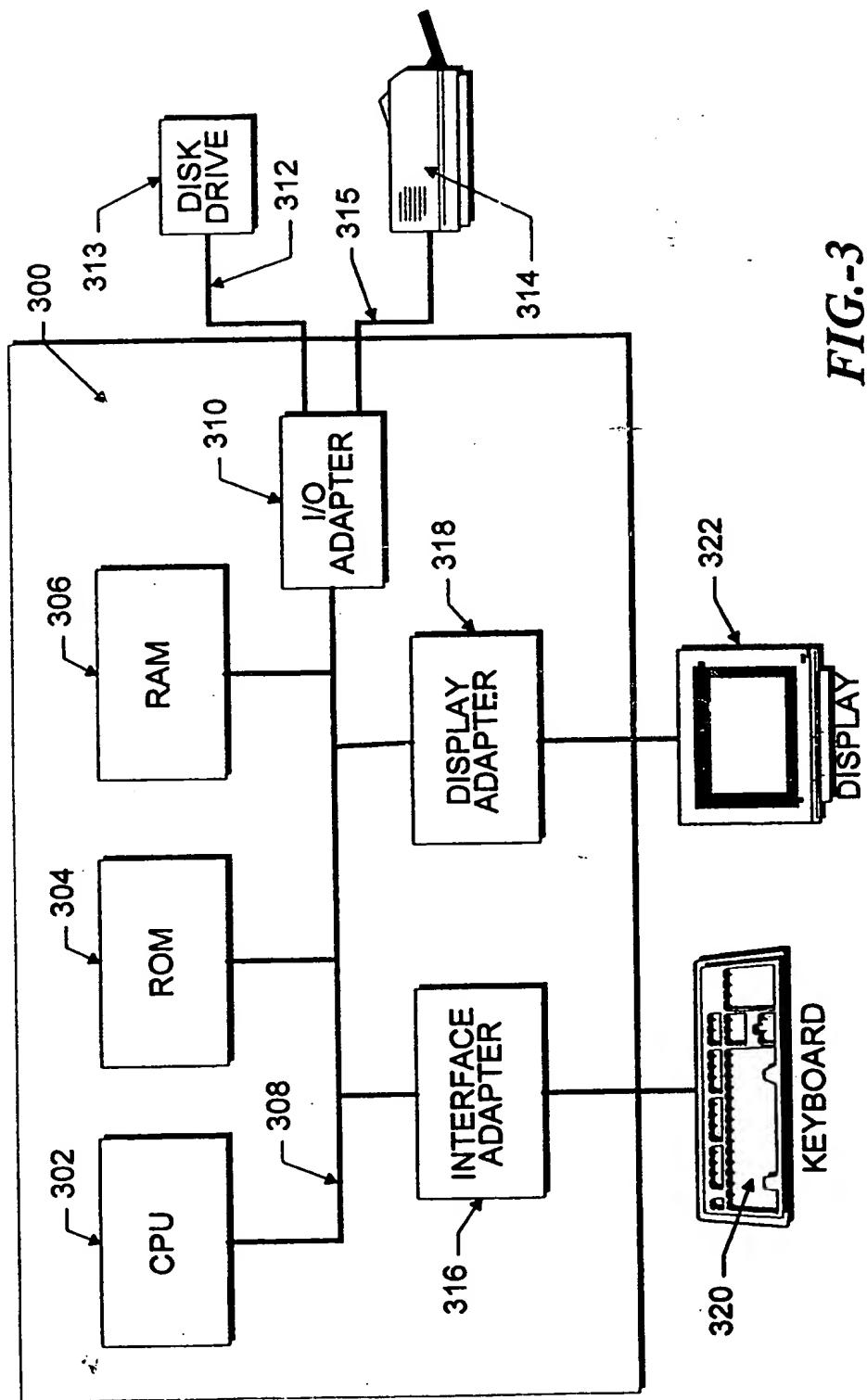
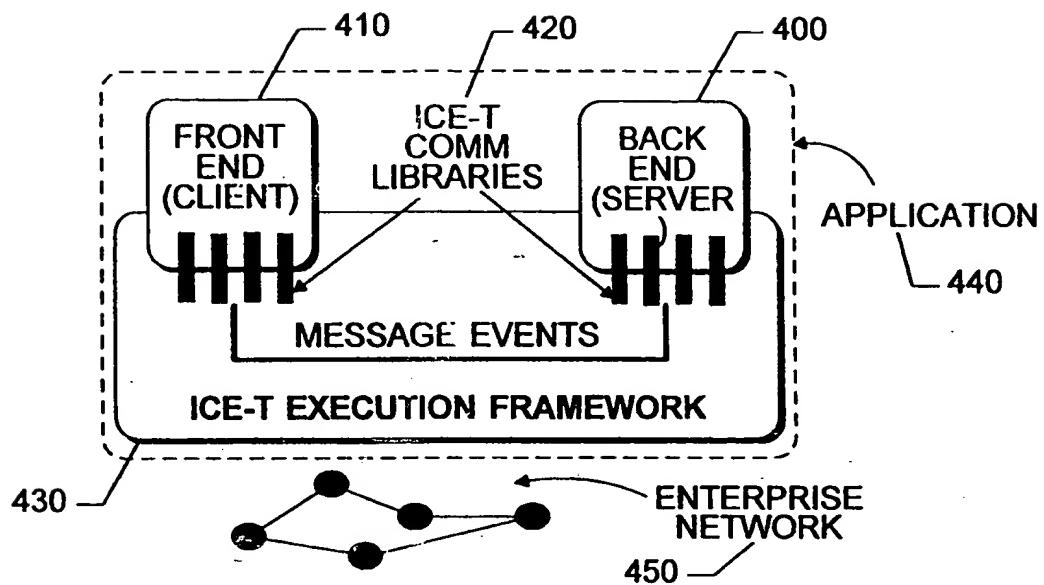
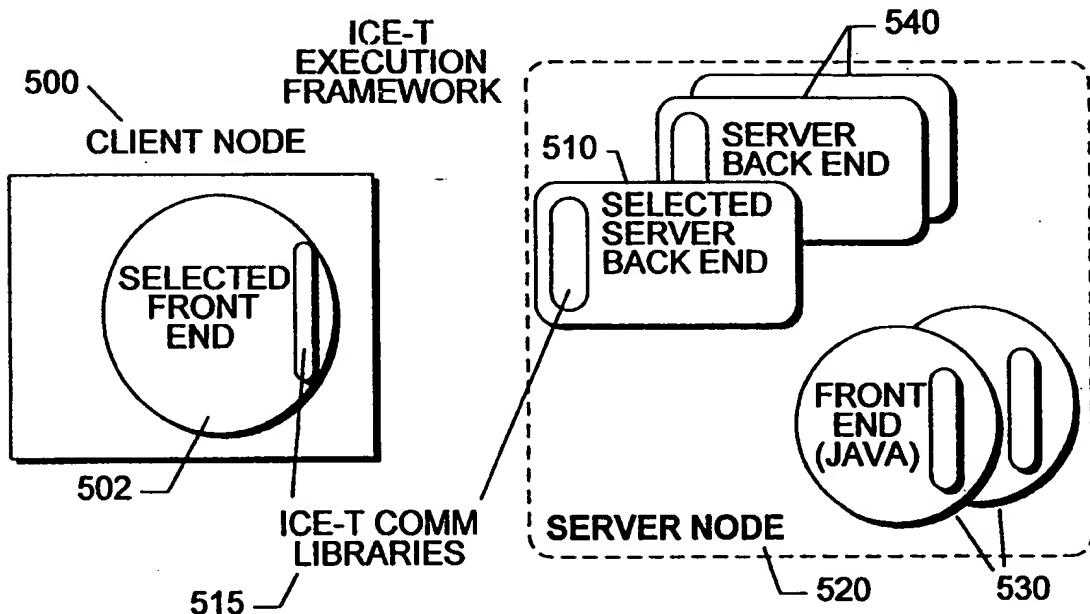


FIG.-3

**FIG.-4****FIG.-5**

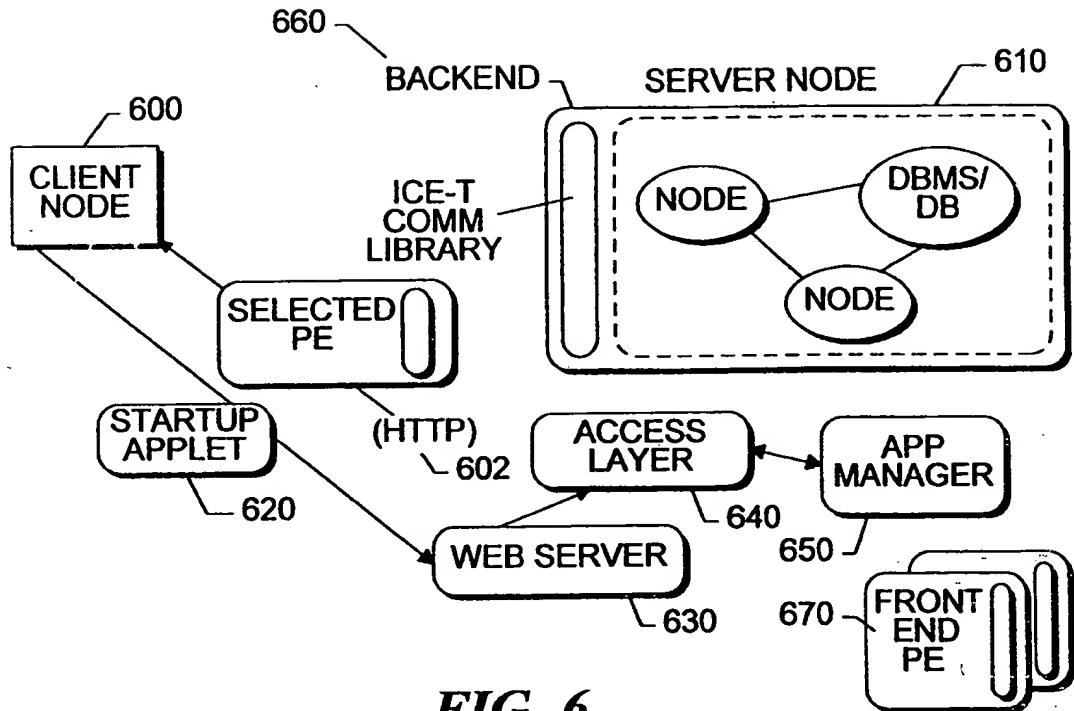


FIG.-6

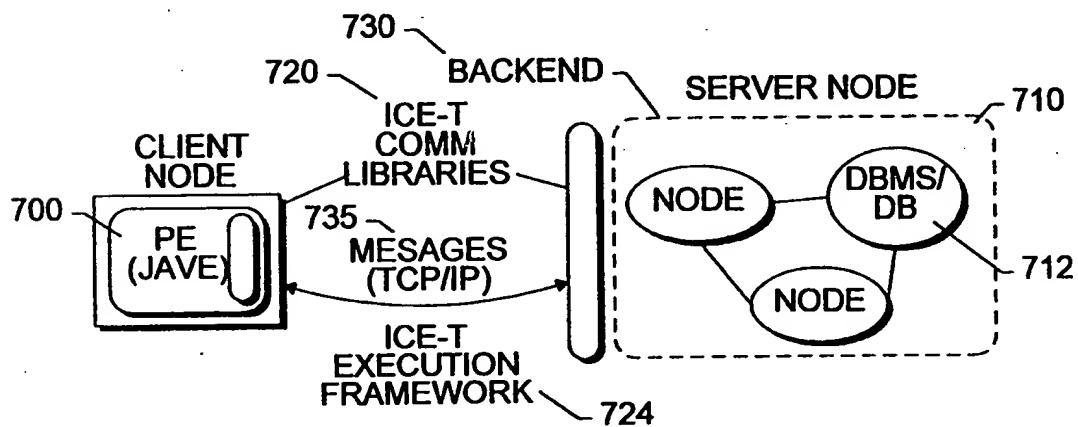
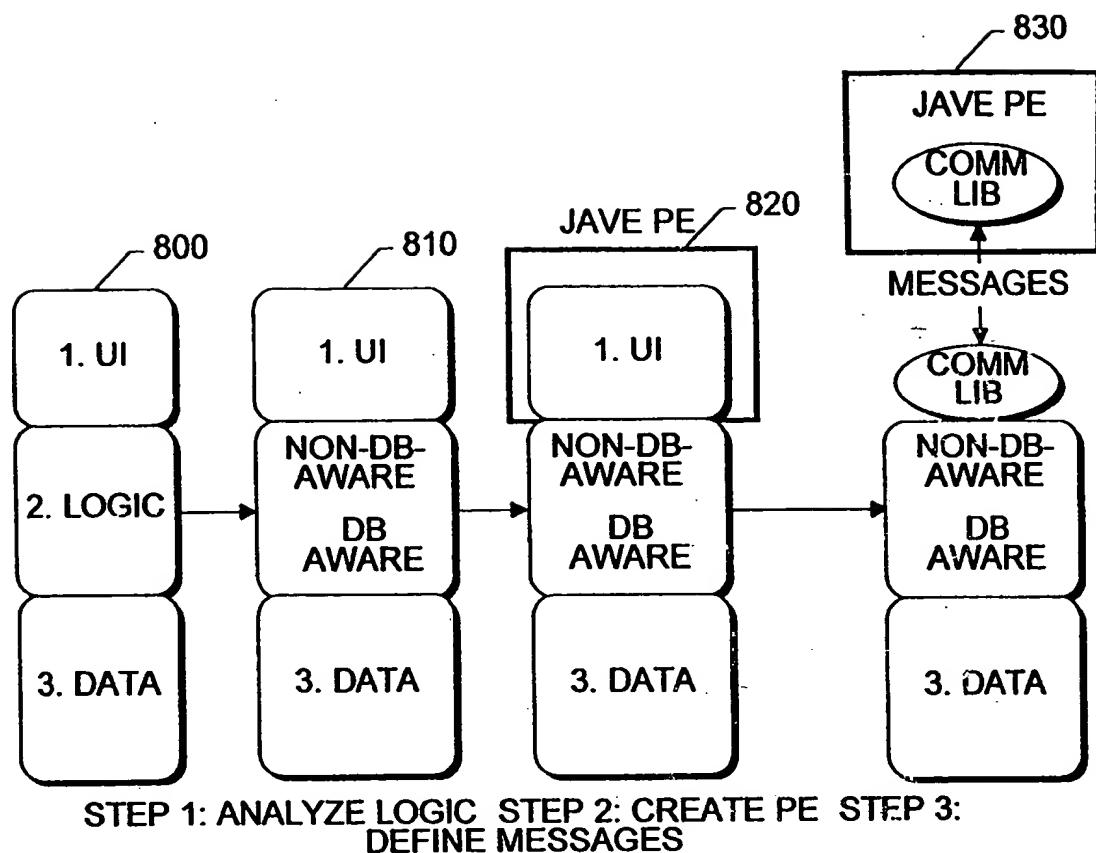
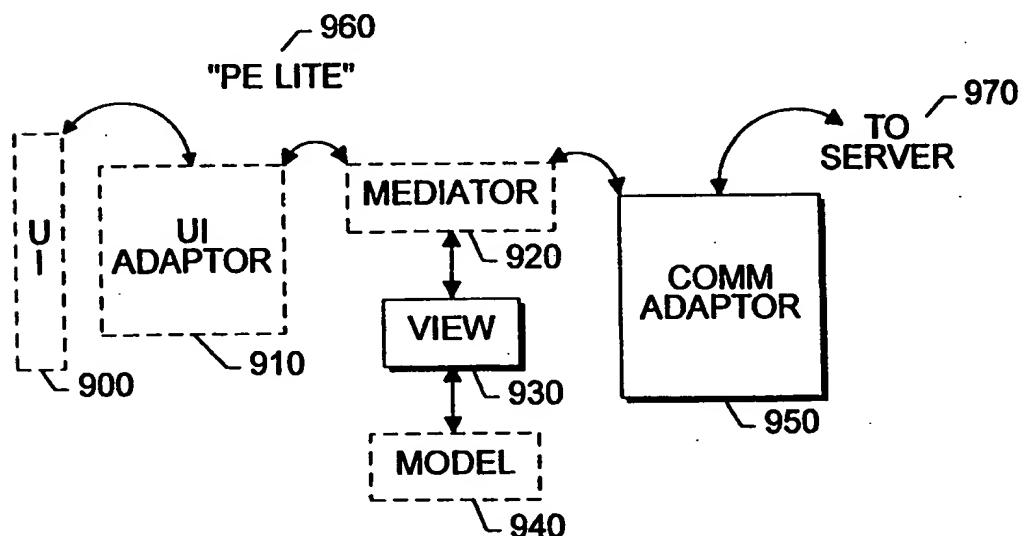


FIG.-7

**FIG.-8****FIG.-9**

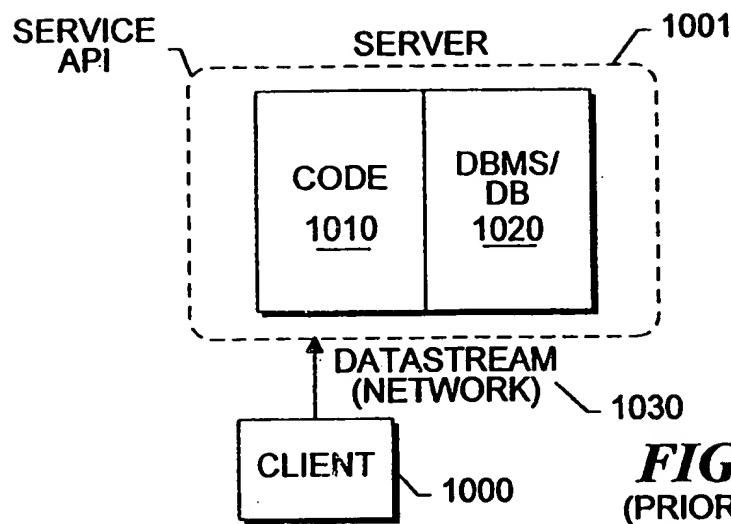


FIG.-10
(PRIOR ART)

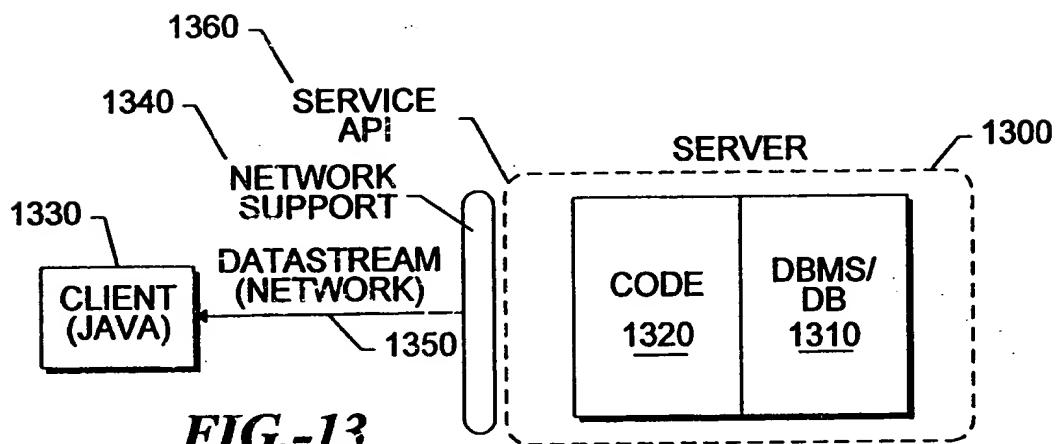


FIG.-13

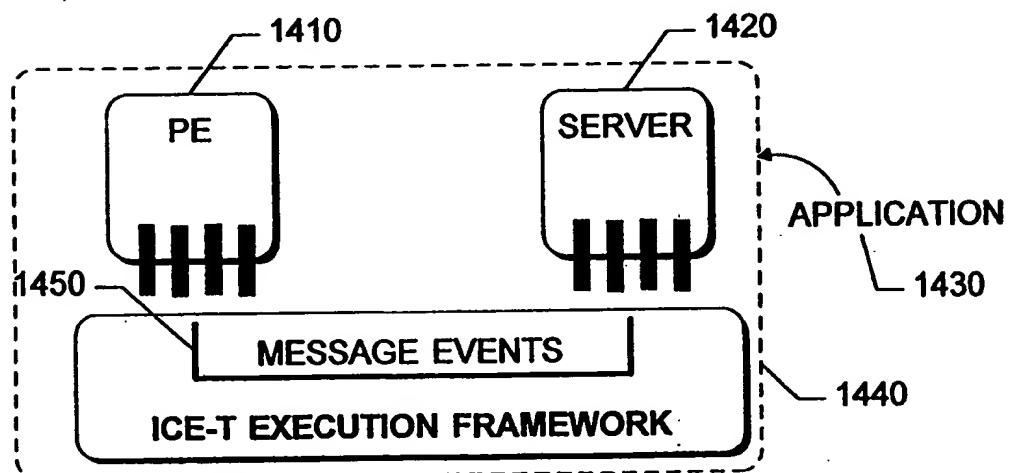


FIG.-14

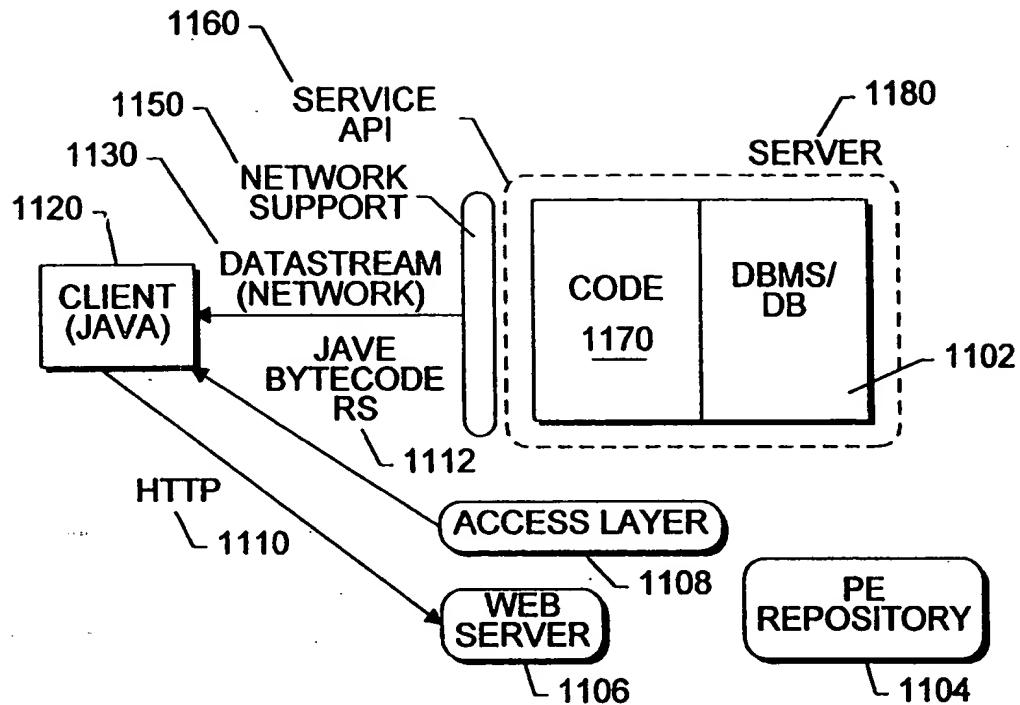


FIG.-11

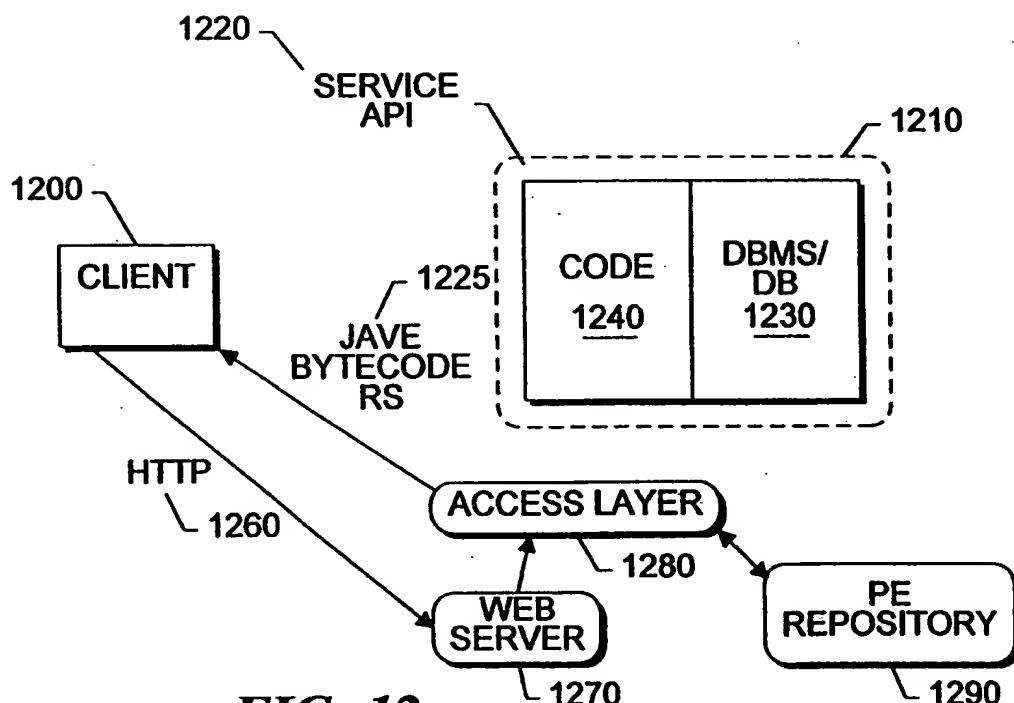


FIG.-12

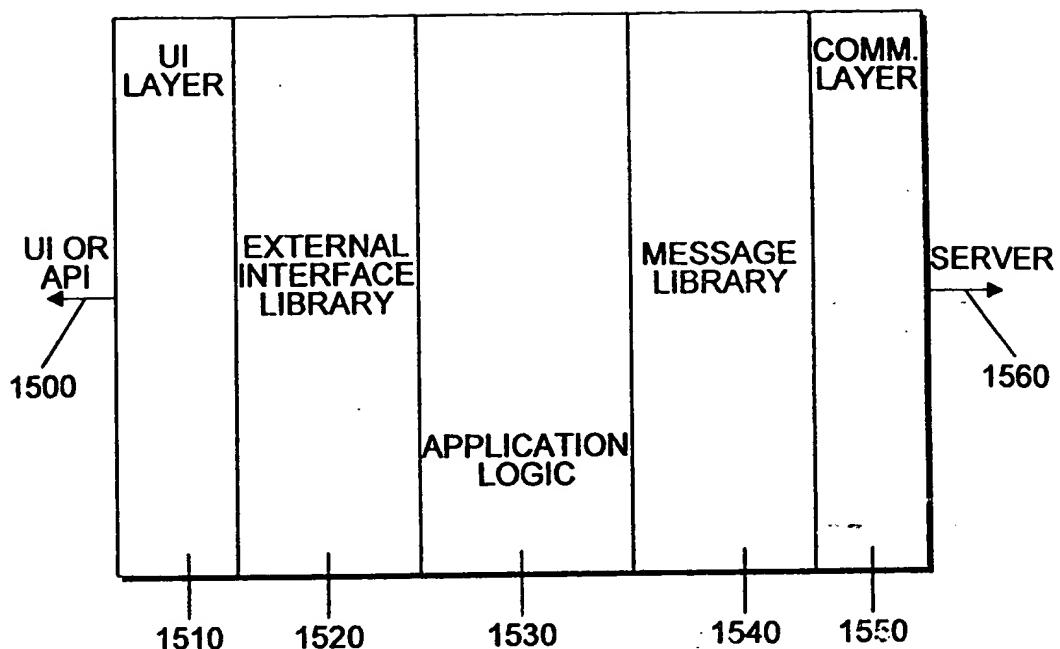


FIG.-15

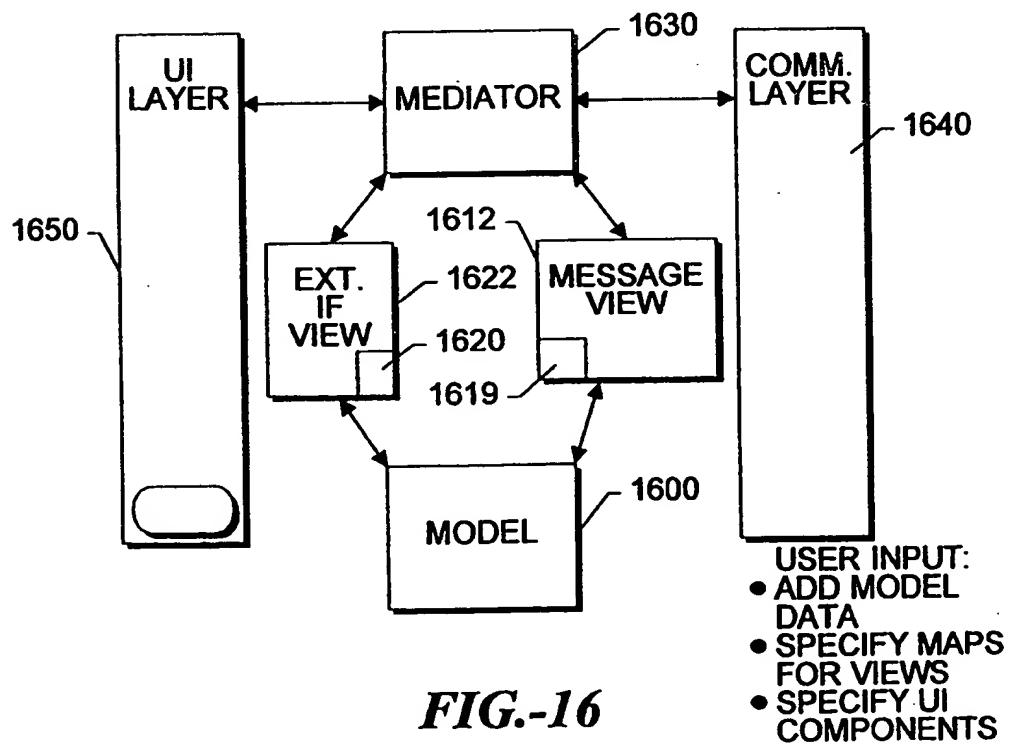
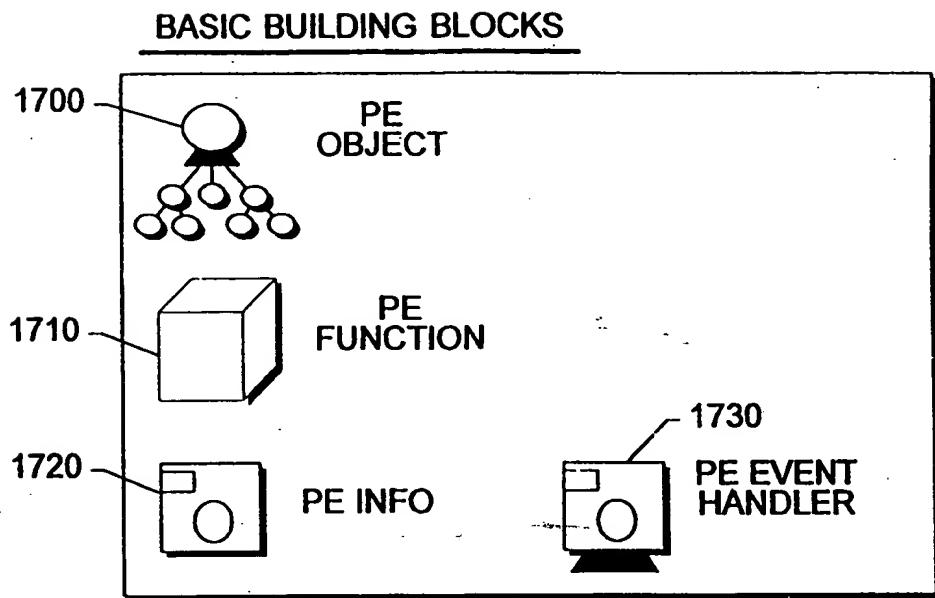
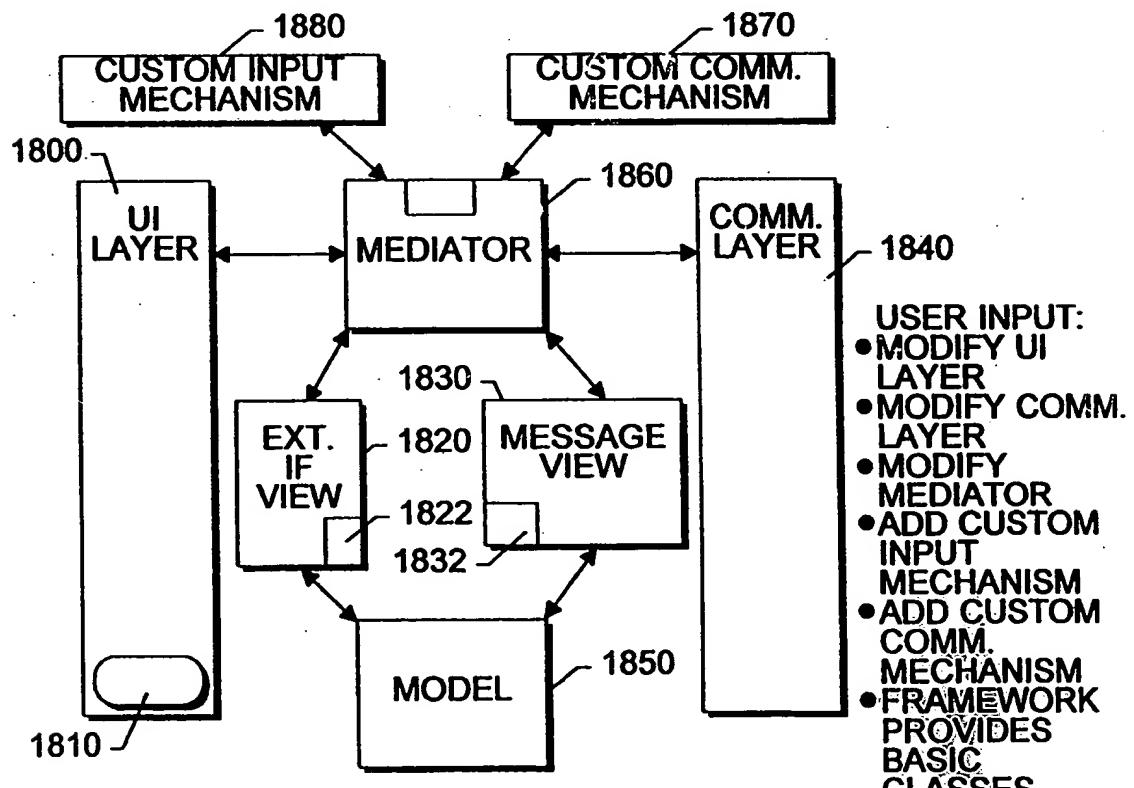
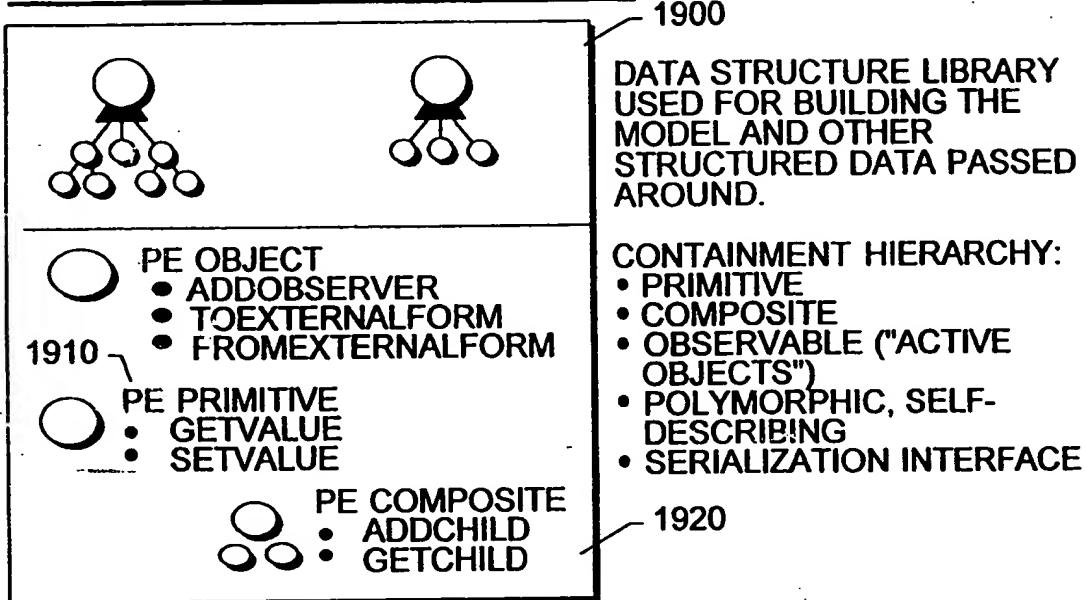
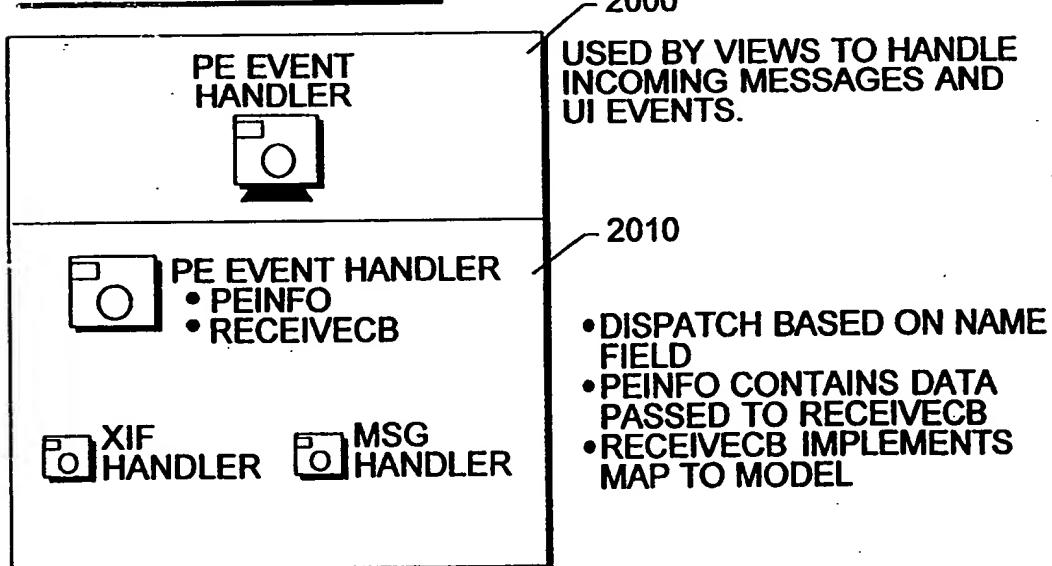
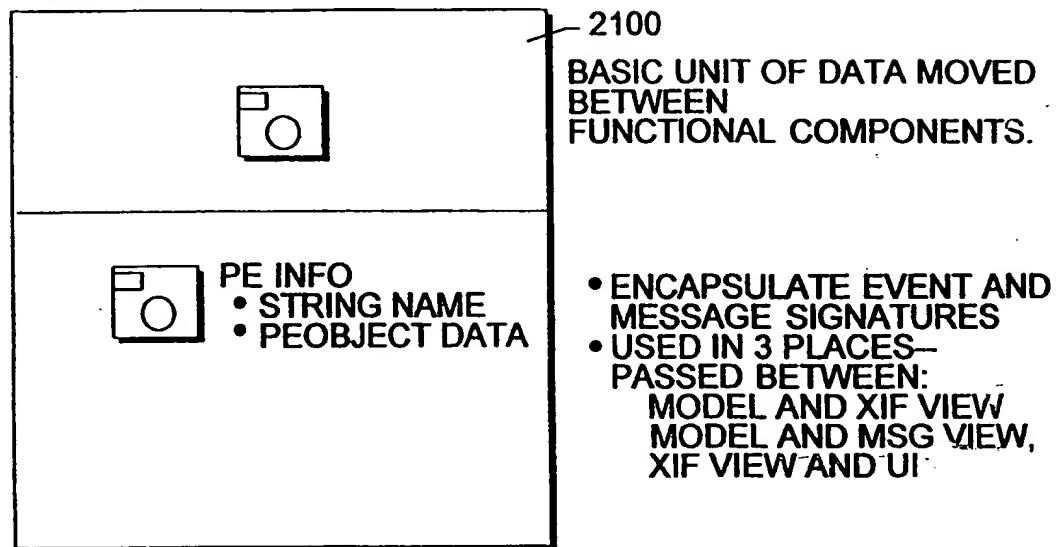
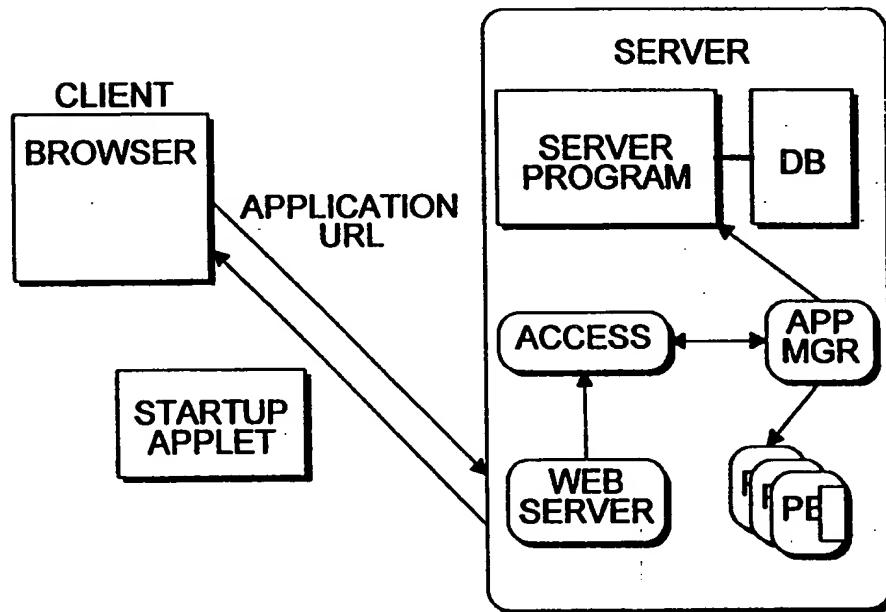


FIG.-16

**FIG.-17****PE DEVELOPMENT: EXTENDING THE FRAMEWORK****FIG.-18**

BASIC BUILDING BLOCKS: PE OBJECT**FIG.-19**BASIC BUILDING BLOCKS**FIG.-20**

BASIC BUILDING BLOCKS: PE INFO**FIG.-21****FIG.-26**

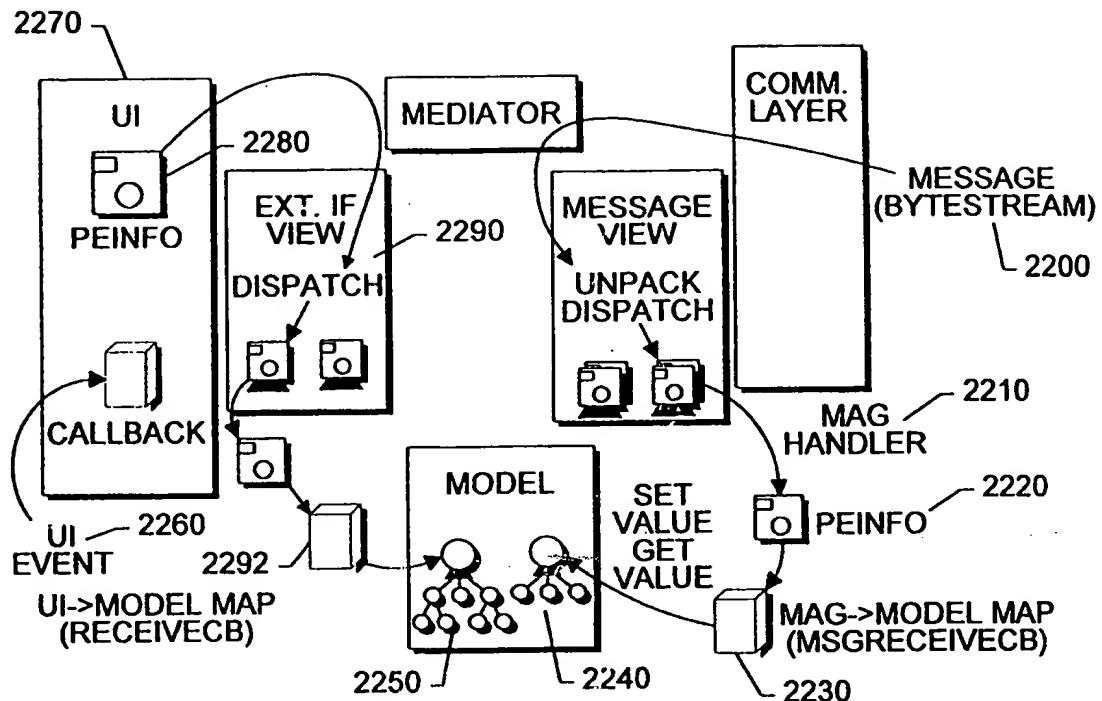


FIG.-22

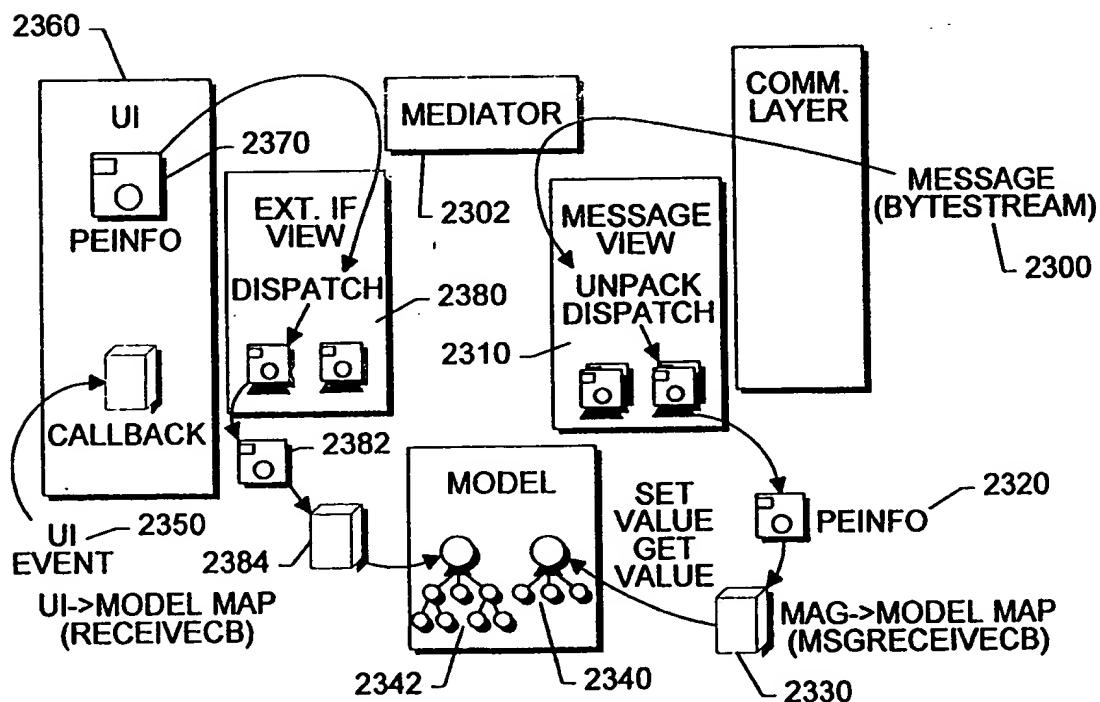
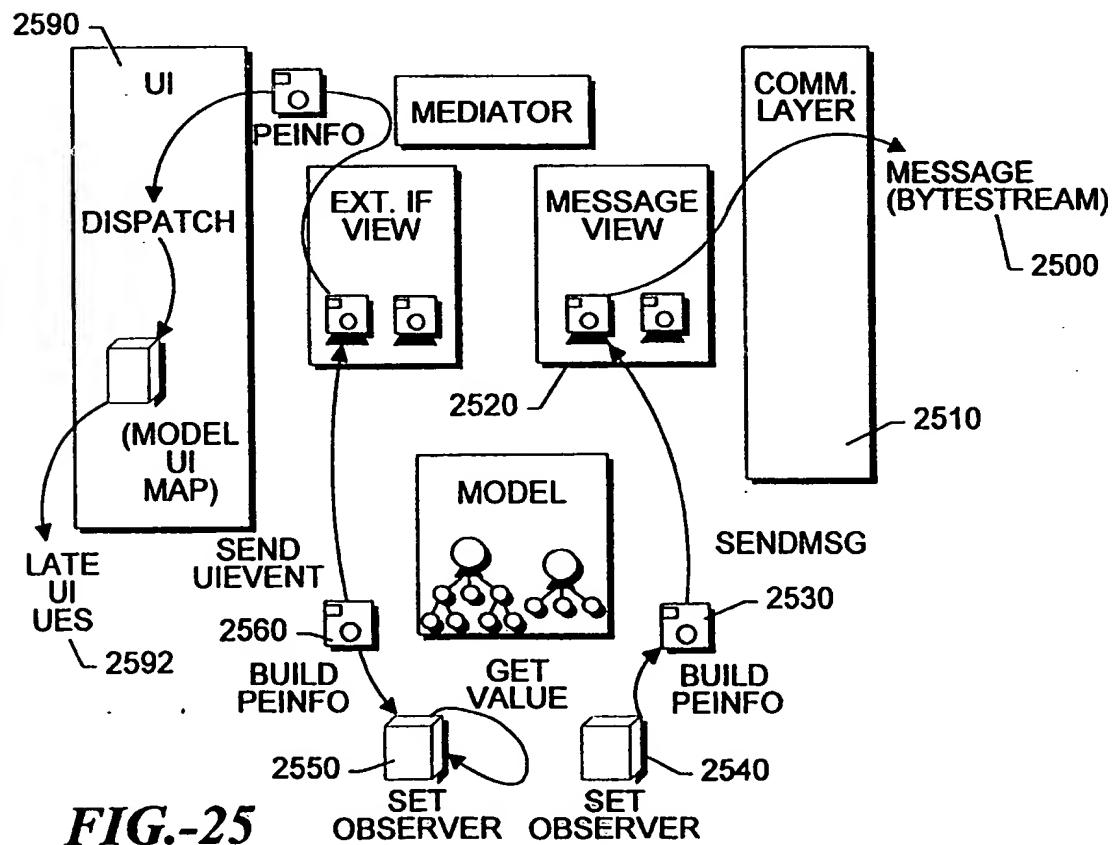
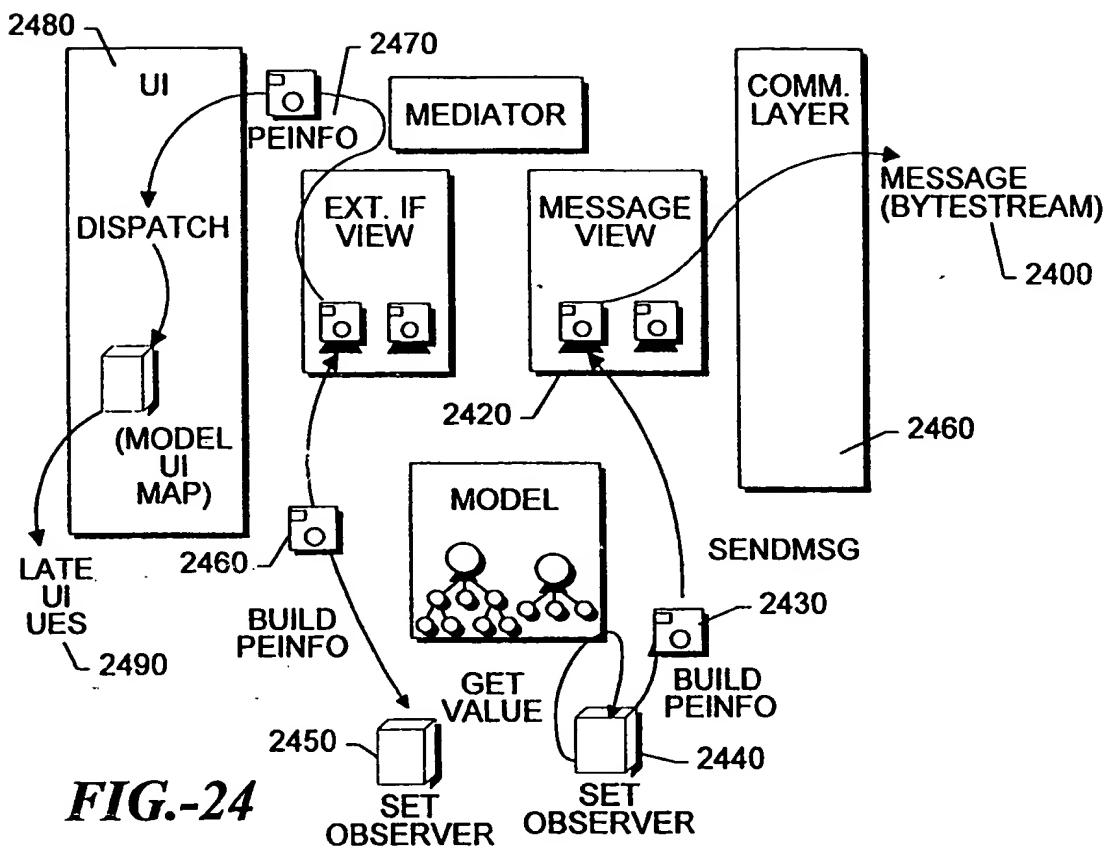


FIG.-23



NAME	DESCRIPTION	DEVELOPMENT	RUNTIME
PRESENTATION ENGINE CLASS	IS THE SUPERCLASS FOR PRESENTATION ENGINES.	IS DERIVED FROM. IS NOT MODIFIED. IS MODIFIED.	IS USED BY YOUR INSTANCE OF THE PRESENTATION ENGINE.
PE-TEMPLATE	IS AN INSTANCE OF THE PRESENTATION ENGINE CLASS.	DEVELOPERS COPY THIS FILE. GIVE IT A NAME AND WRITE METHOD BODIES FOR A FEW METHODS	
MY PRESENTATION ENGINE	(A DEVELOPED AND COMPILED PRESENTATION ENGINE, GIVE AN EXAMPLE HERE)	IS DEVELOPED FROM THE PE-TEMPLATE. IS AN INSTANCE OF THE PRESENTATION ENGINE CLASS, CREATED USING PE-TEMPLATE AND FILLED IN WITH APPLICATION-SPECIFIC CODE.	A COMPILED PRESENTATION ENGINE (HERE NAMED MY PRESENTATION ENGINE) THAT INCLUDES A GUI AND AN INSTANCE OF THE CLIENT COMMUNICATION LIBRARY.

FIG.-27

FUNCTION	DESCRIPTION	REQUIREMENTS
SET SERVER COMMUNICATION PROPERTIES ()	SETS SERVER COMMUNICATION LIBRARY PROPERTIES SUCH AS SERVER TIMEOUTS, AND DEBUG FILE AND TRACE LEVEL.	SHOULD NOT WRITE ANYTHING TO STDOUT BECAUSE THE ICE-T ACCESS LAYER EXPECTS THE FIRST THING WRITTEN TO STDOUT TO BE THE PORT NUMBER THAT THE SERVER USES TO LISTEN FOR THE CLIENT CONNECTION
CREATE MESSAGE HANDLERS()	REGISTERS THE MSG HANDLERS, IS THE SRVR ANALOG TO THE CREATE MSG HANDLERS () METHOD IN THE PRESENTATION ENGINE.	CALLED IMMEDIATELY AFTER THE SET SERVER COMMUNICATION PROPERTIES () CALL

FIG.-28

PROPERTY	DEFAULT ACCEPTED VALUES	
SET SERVER TIMEOUT()	3600 SECONDS (1 HOUR)	>0 SPECIFYING 0 OR <0 MEANS NO TIMEOUT
SET DGB FILE	STDERR	USER-SPECIFIED FILENAME
SET ACCEPT TIMEOUT	3600 SECONDS	>0 SPECIFYING 0 OR <0 MEANS NO TIMEOUT
SET TRACE LEVEL	0 (ZERO)	0 MEANS NO TRACE MESSAGES ARE PRINTED. 2 MEANS TRACE MESSAGES ARE PRINTED.
SET THREAD MODEL	SERVICE COMMUNICATION-THREAD-MODEL-SINGLE-HANDLER	SRVCOb-1-THREAD-MODEL-SINGLE-HANDLER SRVOI-THREAD-MODEL-MULTIPLE-HANDLERS

FIG.-29

TABLE D-1 DESCRIBES CLIENT AND SERVER-SIDE EXCEPTIONS
TABLE D-1 ICE-T EXCEPTIONS

EXCEPTION	DESCRIPTION
EXCEPTIONS THROWN BY THE CLIENT PROI RAM (JAVA CODE):	
PE EXCEPTION	A GENERAL EXCEPTION THAT SIGNIFIES AN UNIDENTIFIED PROBLEM. THE MESSAGE MAY INCLUDE MORE DETAILS ABOUT WHERE IT WAS RAISED. PE~XCEP~ION IS USED VERY RARELY.
MAP NOT FOUND EXCEPTION	RAISED WHEN A MODULE IN THE PRESENTATION ENGINE RECEIVES A MESSAGE FOR WHICH NO MAP FUNCTION HAS BEEN REGISTERED. THE MESSAGE EXPLAINS WHERE AND WHY THE EXCEPTION WAS RAISED.
DUPLICATE MAP EXCEPTION	RAISED WHEN THE CLIENT PROGRAM ATTEMPTS TO REGISTER THE SAME MESSAGE IN BOTH THE UI AND MODEL.
CCMM EXCEPTION	A GENERAL COMM. EXCEPTION. IT IS INDICATES THAT THERE HAS BEEN SOME PROBLEM AND PRINTS A MESSAGE WITH MORE DETAILS.
CONNECTION EXCEPTION	INDICATES THAT THERE WERE SOME PROBS IN CONNECTION TO THE SERVER.
ICE-T PROTOCOL THAT	PEPROTOCOL EXCEPT. INDICATES THAT THERE WAS A PROB. WITH THE IS BEING SENT BETWEEN THE CLIENT AND SERVER.
EXCEPTIONS THROWN BY THE SERVER COMMUNICATION LIBRARY:	
SERVER EXCEPTION	THE BASE CLASS OF THE OTHER EXCEPTIONS. NEVER THROWN.
SERVER COMMUNICATION EXCEPTION	A GENERAL COMM. EXCEPTION. INDICATES THAT THERE HAS BEEN SOME COMM. PROBLEM AND PRINTS A MESSAGE WITH MORE DETAILS
SERVER CONNECTION EXCEPTION	INDICATES THAT THERE IS A PROB. WITH THE CONNECTION TO THE CLIENT. IT USUALLY INDICATES THAT EITHER THE CLIENT HAS SUT DOWN UNEXPECTEDLY OR THE CONNECTION HAS TERMINATED, AND THE SRVR IS SOMEHOW CONTIN. ON.
SERVER PROTOCOL EXCEPTION	INDICATES THAT THERE HAS BEEN A PROBLEM WITH THE MSG. SENT TO, OR RECEIVED FROM, THE CLIENT. EITHER THE MSG. TYPE IS UNKNOWN, OR THERE HAS BEEN A PROBLEM MARSHALLING OR UNMARSHALLING THE DATA. IT MEANS THAT SOMEHOW THE BYTES THAT WERE SENT ON THE SOCKET WERE GARBLED.

FIG.-30

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include but are not limited to the items checked:

- BLACK BORDERS**
- IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- FADED TEXT OR DRAWING**
- BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- SKEWED/SLATED IMAGES**
- COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- GRAY SCALE DOCUMENTS**
- LINES OR MARKS ON ORIGINAL DOCUMENT**
- REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- OTHER :** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents *will not* correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.